

University of Warwick institutional repository: <http://go.warwick.ac.uk/wrap>

A Thesis Submitted for the Degree of PhD at the University of Warwick

<http://go.warwick.ac.uk/wrap/56939>

This thesis is made available online and is protected by original copyright.

Please scroll down to view the document itself.

Please refer to the repository record for this item for information to help you to cite it. Our policy information is available from the repository home page.

Automatic Coset Systems

Ian Douglas Redfern

A thesis submitted for the degree of Doctor of Philosophy
to the University of Warwick

based on research conducted in the
Mathematics Institute, University of Warwick

September 1993

Table of Contents

1.	Introduction and Definitions	1
2.	Change of Generators and Extensions	8
3.	Simple Construction of Coset Automata	14
4.	Coset Enumeration by Depth-first Search	20
5.	An example – Punctured Tori	23
6.	Knuth–Bendix	29
7.	Automaton Induction	36
8.	Multipliers	44
9.	Example	46
10.	Existence	54

Appendix 1: Results

Appendix 2: Programs

Bibliography

Table of Illustrations

Word Rejector	16
Coset Word Rejector	17
Free Group Word Acceptor	20
Hausdorff Dimension Graph	28
Induction Example	36
Truncation Example	38
Hyperbolic Space	54
Extended Rips Condition	56
Early Take-off	57
Parallelism	57
Quasiconvexity	61
Tetrahedron Finite Rule Set	A1
Tetrahedron Both Parts	A1
Punctured Torus: $p/q = 1/11$	A1
Punctured Torus: $p/q = 13/34$	A1
Square	A1
Square Markov Partitioned	A1
Hexagon	A1
Klein–Fricke 1	A1
Klein–Fricke 1 Thin States	A1
Klein–Fricke 2	A1
Klein–Fricke 2 Zoomed Out	A1
Klein–Fricke 2 Markov Partitioned Zoomed Out	A1

Automatic Coset Systems

Summary

This thesis describes the theory of automatic coset systems. These provide a simple and economical way of describing a system of cosets in a group with respect to a subgroup, such as the cosets of the stabiliser of an object under a group of transformations.

An automatic coset system possesses a finite state automaton that provides a name for each coset, and a set of finite state automata that allow these cosets to be multiplied by group generators.

An algorithm is given that will produce a certain type of automatic coset system, should one exist, from a description of the group and subgroup. The type of system produced has the advantage that it names each coset uniquely using as short a name as possible. This makes it particularly useful for coset enumeration, and several examples of its use are given in an appendix.

Two theorems are also proved: the property of being an automatic coset system is independent of the generating set chosen, and quasiconvex subgroups of hyperbolic groups have automatic coset systems.

Acknowledgements

The examples of automatic coset systems in this thesis were given to me by Oliver Goodman of the Geometry Center at the University of Minnesota, and Greg McShane, John Parker, André Rocha and Derek Holt of the University of Warwick. The systems were computed using machines belonging to University of Warwick Computing Services and the Geometry Center. The Geometry Center also provided me with the opportunity to spend three weeks in May 1993 in Minneapolis, during which I had the chance to talk to several expert geometers and collect useful examples, as well as work on the programs. Throughout my three years at Warwick I have been supported by the SERC. I should like to thank the staff of University of Warwick Computing Services for the facilities they provided that made this work possible. I should also like to thank Professor David Epstein of the University of Warwick for several long and useful talks. I must also thank my supervisor, Derek Holt, without whom none of this would have happened, for invariably nudging me in the right direction, even if this involved a 180 degree turn. Finally, many thanks and much pizza to A, A, 1, 14 and 103, who collectively got me through.

Declaration

The work in Chapter 5 on punctured tori is joint research with John Parker and Greg McShane of the University of Warwick and is to be published separately. The rest of this thesis was written by me, and the work contained in it is my own except where otherwise stated.

Chapter 1: Introduction and Definitions

Aim

This chapter briefly describes the contents of the rest of this thesis and defines the fundamental objects of study.

Concepts and Intentions

Fundamental to this work is the concept of the *finite state automaton*, which is a computing device suited to the fast processing of long strings of symbols, which is something of great value in Group Theory. This thesis builds upon the work in [EHR] and [CEHLPT] which uses finite state automata to do calculations in groups, both for enumerating elements and multiplying words.

This thesis expands this work to the less structured area of coset systems, where we now want to describe and multiply words relative to a subgroup. It is particularly suited to describing the orbit of an object under a group action: we turn this into a problem inside the group by looking at the cosets of the stabiliser of the object, and then can enumerate the points in the orbit by enumerating the cosets.

Much of the earlier work does not go through to the new situation, and that which does often has to be modified to construct automata explicitly. In particular, the algorithm used for constructing the automata for automatic groups does not work for automatic cosets and a different, more general algorithm is necessary.

In this chapter, I define an automatic coset system, and in Chapter 2 I show that the definition is independent of the choice of generating set. In Chapter 3, I show how an automatic coset system can be made in a simple case, such as a free group, and in Chapter 4 I show how to use the automatic system for coset enumeration. The simple case is itself extremely useful, as is shown in Chapter 5 where it is used to construct drawings of circle packings.

The next three chapters are devoted to constructing automatic coset systems in the more general case, where a more complicated guessing algorithm has to be used, and Chapter 9 contains a worked example.

Finally, in Chapter 10, the existence of automatic coset systems is shown in one quite useful situation: quasiconvex subgroups of hyperbolic groups. The appendices contain example coset automatic systems and sample code from the programs that produce them.

Definitions

The notation and basic definitions will follow [CEHLPT], and most of the basic theorems in there will just be quoted without proof, to avoid needless duplication. The following definitions are included for the sake of fixing notation.

An *alphabet* A is a set, with elements called *letters*. A *string* over A is a finite (possibly empty) sequence of letters; the length of a string u is written $|u|$ and is its length as a sequence. The set of all strings over A is written A^* , and the empty string (which has length 0) is written ϵ . A *language* over A is a subset of A^* . We can consider A^* to be a semigroup under the action of concatenation (written, where necessary, as $u_1 \cdot u_2$). If t is a natural number then $u(t)$ is the prefix of u of length t , or the whole of u if $t \geq |u|$.

If A is an ordered alphabet, we say that, for two strings $u, v \in A^*$ of the same length, u is *lexicographically less than* v if

$$u = u_1 \dots u_n$$

$$v = u_1 \dots u_k v_{k+1} \dots v_n$$

$$u_{k+1} < v_{k+1} \text{ in } A$$

i.e. we compare the letters in u and v at the first position in which they differ. For strings of different lengths, we adopt the convention that any prefix of u is lexicographically less than u ; otherwise we again look at where u and v first differ.

We define the *Short-Lex* ordering on A^* as follows: For $u \neq v \in A^*$, if u is shorter than v then $u < v$. If they are the same length and u is lexicographically less than v then $u < v$. Otherwise, $v < u$. This is a well-ordering on A^* ; that is, in any collection of strings there is always a least one, which is also one of the shortest.

Languages and Finite State Automata

A *deterministic finite state automaton* is a quintuple (S, A, μ, Y, s_0) , where S is a finite set whose elements are called *states*, A is a finite alphabet, $\mu: S \times A \rightarrow S$ is a function, called the *transition function*, Y is a subset of S , whose members are called the *accept states* and $s_0 \in S$ is called the *start state*. A *finite state automaton* will mean a deterministic one unless explicitly marked otherwise.

The idea is that the automaton is a model of a very simple machine, which starts in the start state s_0 , and is fed letters, one at a time. After reading a letter, the internal state of the machine is changed to a possibly new state,

determined by its current state, the letter read and the transition function μ . If, after consuming all the input, the state of the machine is in Y , the machine answers ‘Yes’ and accepts the input; otherwise it answers ‘No’ and rejects it.

Formally, if $M = (S, A, \mu, Y, s_0)$ is a finite state automaton, let $\text{Map}(S, S)$ be the semigroup of maps from S to itself. The map $\hat{\mu}: A \rightarrow \text{Map}(S, S)$ induced from μ defined by $\hat{\mu}(a)(s) = \mu(a, s)$ can be extended to a semigroup homomorphism μ^* from A^* to $\text{Map}(S, S)$. A string w is accepted by M if $\mu^*(w)(s_0) \in Y$, and the language of M , written $L(M)$, is the set of all such acceptable strings. Note, in particular, that ϵ corresponds to the identity in $\text{Map}(S, S)$.

We can represent a finite state automaton as a directed graph, with a node for each state and an edge (or arrow) for each transition.

It is also useful to work with *non-deterministic* automata where states may have more than one arrow labelled with a given letter leaving them, and where ‘empty’ arrows, labelled by the empty string ϵ , are allowed. Here, at any state we have the choice of reading input or following an ϵ -arrow, and if we do read input we may follow any of the arrows out that are labelled with that letter.

The languages acceptable by this larger class of machines are exactly the same as those acceptable by deterministic ones, and so to any non-deterministic automaton there corresponds at least one deterministic one that accepts the same language; producing such a deterministic automaton is called *determinising* the non-deterministic automaton and can result in an exponential increase in the number of states. See [AHU] Chapter 9 and [CEHLPT] Chapter 1 for details.

A further class of languages are those that can be described by a *regular expression*. A regular expression is built up recursively out of the letters in A and the symbols ϵ , $(\)$, \cdot , $|$ and $*$. A string is said to *match* a regular expression if it is in its language; the letters in A and ϵ match their corresponding regular expressions.

If R and S are regular expressions, then (R) is matched by the same set of strings as R is; $R \cdot S$ is matched by any concatenation of a string matching R with a string matching S ; $R|S$ is matched by any string matching R or S or both; and R^* is matched by the concatenation of any finite number (including zero) of strings that match R .

In fact, the class of languages matching regular expressions is the same as that acceptable to finite state automata, and so to each regular expression there corresponds at least one automaton, and vice versa. See [AHU] Chapter 9 or [CEHLPT] Chapter 1 for details. Sometimes it can be convenient to construct automata via regular expressions, and this technique will be used in following

chapters. For details on the process (which is best done by an intermediate non-deterministic automaton) see [AHU] or [CEHLPT].

It is further true that the above class of languages is closed under operations of predicate calculus, including \neg , \wedge , \vee , \exists and \forall . The latter two are used when we are dealing with automata over alphabets whose elements are pairs or n -tuples of elements; these are known as *n-tape automata*. The predicate calculus operations allow us to make, for example, a new automaton that accepts the set of u such that there is some v with (u, v) acceptable to our original automaton; this process is called *projection*.

Sometimes when dealing with two-tape automata we want to have them accept pairs of strings of differing lengths; to do this we add a *padding* or *end-of-string* letter, '\$', to the end of the shorter string as often as necessary. We will occasionally need to add such a letter to an ordinary automaton W to produce an automaton $W_\$$ which accepts the language $L(W) \cdot \* ; this is called the *padded* version of W , and is most often used to describe the projections of two-tape automata.

Automata and Group Theory

Let G be a group and A an alphabet, with $\pi: A^* \rightarrow G$ a semigroup homomorphism from A to G . If $w \in A^*$ we say $\bar{w} = \pi(w)$ is the element of G *represented* by w . If π is surjective, we say A *generates* G *as a semigroup* and that A is *a set of semigroup generators for* G .

Sometimes we shall add an end-of-string symbol, '\$'; we shall then take $\pi(\$) = 1$, the identity of G . Normally, though not necessarily, for each letter in A there will be another letter that corresponds to its inverse; we then say A is *inverse-closed*. If A is inverse-closed we often denote the inverses using capital letters.

We are now in a position to define an automatic coset system:

Let G be a finitely generated group and let H be a subgroup of G . An *automatic coset system* for G/H consists of a finite set of semigroup generators A for G , a finite state automaton W over A and a set of automata M_x over $(A \cup \{\$\}) \times (A \cup \{\$\})$ for each $x \in A \cup \{\epsilon\}$, such that:

$\pi(L(W))$ contains at least one member of each coset G/H and

For each $x \in A \cup \{\epsilon\}$, $(w_1, w_2) \in L(M_x)$ if and only if $w_1, w_2 \in L(W) \cdot \* and $H\bar{w}_1x = H\bar{w}_2$.

We call W the *word acceptor*, M_ϵ the *equality recogniser* and each M_a for

$a \in A$, a *multiplier*. Because the two inputs to a multiplier may be of different length, they are padded at the end with the '\$' symbol, which is not normally used in the word acceptor – thus w_1 and w_2 lie in $L(W) \cdot \* (the language of the padded word acceptor, $W_\$$), and not $L(W)$.

If for each coset in G/H the set of strings representing it accepted by the word acceptor is precisely the set of shortest representatives, then the system is called *strongly geodesic automatic*; if it is only a subset of the possible shortest representatives, the system is called *weakly geodesic automatic*. If there is only one representative for each coset and it is the Short-Lex least string that represents it, the system is called *Short-Lex automatic*.

An *automatic group* is a group with an automatic coset system in which $H = 1$. Both concepts are independent of the generating set A of G – this will be proved in Chapter 2.

Note that this definition is not completely general, since we require that A generate G . It is quite possible to have a looser definition in which H contains a normal subgroup of G and we only have enough generators for G/H – taking the direct product of a group K with C_2 and considering $(K \times C_2)/K$ to only have one generator, for example.

This, however, is not helpful in determining useful things about the coset system, and in general we shall not even bother with cases in which H contains a normal subgroup of G , because it plays no part in the structure of G/H .

For automatic groups we have a *Lipschitz property*: since, from any live state in a multiplier there is a finite path to an accept state, any pair of strings acceptable to a multiplier must stay a bounded distance apart. That is, since M_a has k states, for any $(u, v) \in L(M_a)$ and any t , there are two strings p_t and q_t with $(u(t)p_t, v(t)q_t) \in L(M_a)$ and $|p_t|, |q_t| \leq k - 1$. This means that

$$\overline{u(t)p_t a} = \overline{v(t)q_t}$$

and so

$$|\overline{v(t)}^{-1} \overline{u(t)}| = |\overline{q_t}(\overline{p_t a})^{-1}| \leq 2k - 1$$

where $|g| = \min_{u \in A^*} \{|u| : \overline{u} = g\}$ for $g \in G$.

For automatic coset systems this condition no longer holds: instead we get

$$H\overline{u(t)p_t a} = H\overline{v(t)q_t}$$

and so there is some $h_t \in H$ with

$$|\overline{v(t)}^{-1} h_t \overline{u(t)}| \leq 2k - 1.$$

We have no information about this h_t , and this makes it much more difficult to reason about automatic coset systems than automatic groups.

Much can be done if we assume that h_t is bounded, and this is used in the final chapter. In this case it is possible, but not easy, to adapt the algorithms used to make automatic groups to deal with automatic coset systems. To deal with the general case, however, it is necessary to come up with a new collection of algorithms, which are much harder to work with or to prove correctness for.

Also in automatic groups we have the extremely useful property that there is a general theoretical procedure for constructing the multiplier automata M_a once the word acceptor is known. We take the states of the multiplier to be triples (s_1, s_2, g) where s_1, s_2 are states of W_\S , the padded word acceptor, and $g \in G$ with $|g| \leq 2k + 1$. We set the start state to be $(s_0, s_0, 1_G)$ and make all states of the form (s_1, s_2, \bar{a}) accept states when s_1 and s_2 are accept states of W_\S .

If there is an arrow labelled p from s_1 to s'_1 and another labelled q from s_2 to s'_2 in W_\S , and if $|\bar{q}^{-1}g\bar{p}| \leq 2k + 1$ then we make an arrow labelled (p, q) from (s_1, s_2, g) to $(s'_1, s'_2, \bar{q}^{-1}g\bar{p})$ in M_a . The automaton thus constructed is indeed the multiplier required – more details can be found in [CEHLPT].

This does not hold for automatic coset systems, and the multipliers have to be explicitly constructed in each particular situation. This makes proofs that automatic coset systems exist much more complicated, sometimes insurmountably so, not least because multipliers are very hard to visualise. It is doubly unfortunate because all the applications of automatic cosets so far have been based solely on the word acceptor.

As part of this work I have written programs to find word acceptors and multipliers for Short-Lex automatic coset systems. In Chapter 7 I show that the algorithm will eventually, barring machine limitations, find the automatic coset system should it exist. I have also written three applications of the word acceptors to coset enumeration, and some of the pictures produced by them are included. I have not been able to find a general way to guarantee the correctness of the answers produced unless the group is free, and this appears to be a difficult problem.

A sample of the program source is in an appendix; at 15,000 lines it is too long to include in full. The source is freely available, subject to agreement with the SERC, and comes to about 360Kb. It is written in C++ with a PostScript display interface and is known to run on Sun and Silicon Graphics machines; it should be fairly easy to port it to another Unix system, but it probably requires too much in resources to work on a PC or a low-specification Macintosh. It is

a tribute to the C++ language that the construction of something so large by a single amateur was feasible in about a year, and that the result is fairly robust and fast enough to use.

I should like to thank the National Science and Technology Research Center for Computation and Visualization of Geometric Structures (The Geometry Center) in Minneapolis for the facilities they offered me in May 1993, during which time I greatly improved the program and had the opportunity to discuss it with several of their staff and visitors. I should also like to thank the staff of University of Warwick Computing Services for their patience with my innumerable bug reports, and for providing the services without which much of this thesis would have been impossible.

Chapter 2: Change of Generators and Extensions

Aim

This chapter will show that the property of G/H being an automatic coset system is invariant under change of generators of G , and that it also passes to finite extensions of G . The change of generators proof is based on that in [CEHLPT]. We first cover two simplified cases.

Adding an Identity

We can add a letter e with $\bar{e} = 1$ to our generating set A with no difficulties. We need make no change to the word acceptor, so that any string containing e is rejected. We need another multiplier, M_e , but since e is an identity, M_e will do.

Removing an Identity

For some $e \in A$ with $\bar{e} = 1$, we want to create an automatic coset system with generator set $A_2 = A \setminus \{e\}$. If A_2 is empty, we have $A = \{e\}$ and G is the trivial group, and so the word acceptor that accepts only ϵ together with the equality multiplier that accepts only $(\$, \$)^*$ give us an automatic coset system for G . We may therefore assume that A is nonempty.

To obtain the word acceptor for this system we cannot just delete e whenever it appears in a string in the original word acceptor's language, because the multipliers would get out of step. Instead, we need to ensure that the lengths of accepted strings do not change by more than a fixed amount.

We take some string e' over A_2 with $\bar{e'} = 1$ and with length $m > 0$. (For instance, we could take $x \in A_2$ and $u \in A_2^*$ with $\bar{u} = \bar{x}^{-1}$ and set $e' = x \cdot u$.) We then define the new language of accepted strings as follows: we replace every m th occurrence of e by e' , starting with the first, and delete all others.

Building the Word Acceptor

We shall make a non-deterministic automaton W' which, when determinised, will be the word acceptor. Its states will be pairs of the form (s, u) , where s is a state of W and u is a string in A_2^* of length at most $|e'| - 1$. We shall assume that W accepts padded words (if necessary, we just add a terminal loop of '\$').

The start state will be (s_0, ϵ) where s_0 is the start state of W .

A state (s, u) will be an accept state if and only if $u = \epsilon$ and s is an accept state of W .

If u is a nonempty string, let $u[1]$ be its first letter and $u[2-]$ be the remainder. In the following description, $u \in A_2^*$ with $1 \leq |u| \leq |e'| - 1$.

For any arrow labelled p from s_1 to s_2 in W , where $p \neq e$, the following shall be arrows in W' :

p from (s_1, ϵ) to (s_2, ϵ) ,

$u[1]$ from (s_1, u) to $(s_2, u[2-] \cdot p)$.

For any arrow labelled e from s_1 to s_2 in W , the following shall be arrows in W' :

$e'[1]$ from (s_1, ϵ) to $(s_2, e'[2-])$,

$u[1]$ from (s_1, u) to $(s_2, u[2-])$.

This will give us a new word acceptor W' with precisely the language we are seeking. The first time a letter e would have been read in a word acceptable to W , it will instead read the first letter of e' , and place the rest of e' on a queue. Thereafter, each letter that would be read by W is instead placed on the end of the queue and the first letter on the queue is used instead. Each time W would come across a e , the new word acceptor will add nothing to the end of the queue, but still use the first letter, thus making the queue shorter. When $|e'| - 1$ more e 's would have been read by W , the two word acceptors will be synchronised once more.

Unlike in the automatic groups case, we must construct multipliers as well as a word acceptor; the word acceptor will be any projection of a multiplier.

Description of the New Multiplier

We need to construct a new multiplier M'_a from the old M_a that replaces every m th occurrence of e by e' (starting with the first) and deletes all others. It is easy enough to do this for a word acceptor: just keep a count of the number of e 's. For a multiplier, however, it is more complicated.

Consider a multiplier that accepts the pair of strings $(abcdf, wexey)$, and suppose we are replacing e by $e' = uU$. We then want the new multiplier to accept the pair of strings $(abcdf, wuUxy)$.

We can best describe this by using a non-deterministic automaton on the same set of states as M_a and with two finite queues (or FIFO stacks) L and R , the left and right queues respectively. After M'_a reads (a, w) , it reaches a state

where M_a could read (b, e) . Instead of reading (b, e) , it reads $(b, e'[1])$, where $e'[1]$ is the first letter of e' , and places the rest of e' on the right queue. Thereafter, where M_a would read (p, q) , M'_a will read $(p, \text{Head}(R))$ and set $\text{Tail}(R) = q$. It will now keep the right input stream the correct number of letters behind the left one. In this example, then, M'_a reads (b, u) and puts U on R . At the next state, where M_a would read (c, x) , M'_a will read (c, U) and put x on R .

When another e would be required on the right-hand side, M'_a uses the head of the right queue as before, but does not add a letter to the end. In this manner it will exhaust the queue after $|e'| - 1$ more e 's, and go back to normal operation. In this example, M_a would read (d, e) , so M'_a reads (d, x) and then carries on reading strings in the same way as M_a .

It is thus possible to ensure that neither queue has to hold a string longer than $|e'| - 1$, and hence that we have only added a finite amount of storage capacity to the multiplier, and so we still have a language that can be described by a finite state automaton.

Construction of the New Multiplier

I shall now give a formal construction of the new multiplier. This is not intended for use in practice, but will suffice for a demonstration.

The states of the multiplier M'_a will be triples of the form (s, u, v) , where s is a state of M_a and u and v are strings in A_2^* of length at most $|e'| - 1$.

The start state will be $(s_0, \epsilon, \epsilon)$ where s_0 is the start state of M_a .

A state (s, u, v) will be an accept state if and only if $u = v = \epsilon$ and s is an accept state of M_a .

If u is a nonempty string, let $u[1]$ be its first letter and $u[2-]$ be the remainder. In the following description, $u, v \in A_2^*$ with $1 \leq |u|, |v| \leq |e'| - 1$.

For any arrow labelled (p, q) from s_1 to s_2 in M_a , where neither of p or q is e , the following shall be arrows in M'_a :

- (p, q) from $(s_1, \epsilon, \epsilon)$ to $(s_2, \epsilon, \epsilon)$,
- $(p, v[1])$ from (s_1, ϵ, v) to $(s_2, \epsilon, v[2-] \cdot q)$.
- $(u[1], q)$ from (s_1, u, ϵ) to $(s_2, u[2-] \cdot p, \epsilon)$,
- $(u[1], v[1])$ from (s_1, u, v) to $(s_2, u[2-] \cdot p, v[2-] \cdot q)$.

For any arrow labelled (p, e) from s_1 to s_2 in M_a , where $p \neq e$, the following shall be arrows in M'_a :

$(p, e'[1])$ from $(s_1, \epsilon, \epsilon)$ to $(s_2, \epsilon, e'[2-])$,
 $(p, v[1])$ from (s_1, ϵ, v) to $(s_2, \epsilon, v[2-])$,
 $(u[1], e'[1])$ from (s_1, u, ϵ) to $(s_2, u[2-] \cdot p, e'[2-])$.
 $(u[1], v[1])$ from (s_1, u, v) to $(s_2, u[2-] \cdot p, v[2-])$.

For any arrow labelled (e, q) from s_1 to s_2 in M_a , where $q \neq e$, the following shall be arrows in M'_a :

$(e'[1], q)$ from $(s_1, \epsilon, \epsilon)$ to $(s_2, e'[2-], \epsilon)$,
 $(e'[1], v[1])$ from (s_1, ϵ, v) to $(s_2, e'[2-], v[2-] \cdot q)$.
 $(u[1], q)$ from (s_1, u, ϵ) to $(s_2, u[2-], \epsilon)$.
 $(u[1], v[1])$ from (s_1, u, v) to $(s_2, u[2-], v[2-] \cdot q)$.

For any arrow labelled (e, e) from s_1 to s_2 in M_a , the following shall be arrows in M'_a :

$(e'[1], e'[1])$ from $(s_1, \epsilon, \epsilon)$ to $(s_2, e'[2-], e'[2-])$,
 $(e'[1], v[1])$ from (s_1, ϵ, v) to $(s_2, e'[2-], v[2-])$.
 $(u[1], e'[1])$ from (s_1, u, ϵ) to $(s_2, u[2-], e'[2-])$.
 $(u[1], v[1])$ from (s_1, u, v) to $(s_2, u[2-], v[2-])$.

Justification of Correctness

It is clear that any arrow in M_a gives rise to a family of arrows in M'_a ; it is also true that any arrow in M'_a can be traced back to a unique arrow in M_a . We know from the emptiness of otherwise of the two queues which of the four subcases any arrow in M'_a belongs to; we must merely determine whether either of the pair of letters in the original arrow is e . This can be done by comparing the lengths of the queues before and after the arrow: the queues can only grow or shrink when an e is read.

In the above manner we can determine where in M_a any arrow in M'_a came from. Thus for any pair of strings acceptable to M'_a , we can find the pair of strings that were acceptable to M_a , by following the equivalent arrows in M_a . Similarly, for any pair of strings acceptable to M_a , we can follow the path traced by the equivalent pair in M'_a , by following the equivalent arrows and keeping track of the left and right queues. Moreover, these operations are mutually inverse. There is thus a one-to-one equivalence between $L(M_a)$ and $L(M'_a)$.

It is thus true that we know where pairs acceptable to M'_a came from; we

must now show that they do indeed form the language of an a -multiplier. The easiest way to do this is to show that they are obtained from strings acceptable to M_a by replacing e by e' or ϵ as described earlier. It should be clear from following the arrows round that this is what actually happens: the first time an e would be required, an e' is put on a queue, and this queue is used to delay all letters on that side by a fixed amount. When another e is required, the delay is reduced by 1. Hence the pair of words are equal in the group to the old pair, but are now both acceptable to the new word acceptor.

Change of Generators

Having covered the case where A_2 is obtained from A by removing or adjoining a representative of the identity, we now tackle the case of A_2 arbitrary. We may assume from the above that both A and A_2 contain a representative of the identity. We write each letter in A in its shortest form over A_2 and let c be the length of the longest of these strings. We then pad the other strings with identity elements to make them all have length c .

We now replace each letter in the word acceptor by its equivalent string of length c , and each pair of letters in a multiplier by a pair of strings (or string of pairs) of length c . This clearly preserves regularity. We now need to make new multipliers for the letters of A_2 . Since each of these can also be expressed as a string over A , we can use an expression of the form

$$(\exists w_1, \dots, w_d)((u, w_1) \in L(M_{a_1}) \wedge (w_1, w_2) \in L(M_{a_2}) \wedge \dots \wedge (w_d, v) \in L(M_{a_{d+1}}))$$

where $a_1 \dots a_{d+1}$ is the string for a generator of A_2 written over A .

We thus have achieved a word acceptor and a set of multipliers for A_2 and can change generators at will while preserving the property of being an automatic coset system. Since the choice of generators is unimportant, we shall describe a coset system G/H as being *coset automatic* if there is some automatic coset system for it.

Finite Extensions

Let G/H be an automatic coset system and let G be a subgroup of finite index n in G' . Then G'/H is also coset automatic.

Let $\{t_1, \dots, t_n\}$ be a set of new letters representing a transversal of G'/G , that is

$$G' = \cup_{i=1}^n G \overline{t_i}.$$

If W is the word acceptor for G/H , we shall have W' as our word acceptor for G'/H where

$$L(W') = L(W) \cdot (t_1 | \cdots | t_n)$$

which is clearly a regular language.

To make the multipliers M'_a we first need to determine how right multiplication by a affects the transversal. We must calculate strings w_{ij} and numbers k_{ij} from the following set of equations:

$$w_{ij}t_{k_{ij}} = t_i \cdot a_j$$

for each $a_j \in A \cup \{t_1, \dots, t_n\}$. We also construct the multipliers $M_{w_{ij}}$ for each i and j in a similar manner to the previous section using a large predicate calculus expression.

Now, let us take a general pair of strings that are candidates for being acceptable to M'_{a_j} : $(u_1 \cdot t_p \cdot \$^{r_1}, u_2 \cdot t_q \cdot \$^{r_2})$, where $u_1, u_2 \in A^*$, $r_1, r_2 \geq 1$ and $|u_1| + r_1 = |u_2| + r_2$. We want to determine if $H\overline{u_1 t_p a_j} = H\overline{u_2 t_q}$.

We first run all of the $M_{w_{ij}}$'s in parallel on the pair $(u_1 \$^{r_1}, u_2 \$^{r_2})$; some will accept this and some will not. When we read t_p and t_q we store their values – we then check that what follows is a nonempty string of '\$'s. After the strings are completely read, we only consider the result of $M_{w_{pj}}$. If this accepts the pair we gave it, we then test to see if $q = k_{ij}$; if this is so, we accept the original pair – otherwise we reject it.

This construction will allow us to build all the multipliers for G'/H , and hence gives us an automatic coset system.

The converse theorem (going to finite index subgroups) has not yet been proven: the problem lies in ensuring the correct language for the word acceptor, which may be possible if G is automatic. The similar problem of changing H by a finite amount is much harder, because it involves multiplication on the left, and would probably require the system to be biautomatic, that is, having left multipliers as well as right ones.

Chapter 3: Simple Construction of Coset Automata

Aim

A method is given for building the word acceptor for a Short-Lex coset automatic system in a particularly easy example, together with some examples of its use.

The Example

We wish to find a Short-Lex coset word acceptor for the following group:

$$G = \langle a, b, c, d \mid a^2, b^2, c^2, d^2, (ab)^4, (ac)^3, (ad)^2, (bc)^4, (bd)^4, (cd)^4 \rangle$$

with respect to the subgroup

$$H = \langle a, b, c \rangle.$$

I am indebted to Oliver Goodman of The Geometry Center in Minneapolis for this example: it is an infinite co-volume hyperbolic Coxeter group generated by the reflections in the faces of a particular hyperbolic tetrahedron.

Rewrite Rules for Groups

In order to describe this problem to a computer in a usable form, we turn the above presentation into a set of rewrite rules. A *rewrite rule* $l \Rightarrow r$ is a pair of strings which are equal in the group; any occurrence of the first one can then be rewritten as the second – this is called *reducing* the first string by the rule.

We normally insist that either the second (the right-hand side) be shorter than the first (the left-hand side), or that they be of equal length and that the right-hand one come before the left-hand in dictionary order. This is exactly the same as saying that $l > r$ in Short-Lex order.

We can convert the presentation of G into a set of rewrite rules by making a rule $R \Rightarrow \epsilon$ for each relator R . We get rules of the form $aa \Rightarrow \epsilon$, $adad \Rightarrow \epsilon$ etc. There are many possible equivalent sets of rewrite rules that represent a given set of relations: for example, $da \Rightarrow ad$ is equivalent to $adad \Rightarrow \epsilon$ in the presence of the appropriate inverse generator rules.

We want to produce an automaton that will recognise a unique shortest string for each coset. To do this, we will need some way of determining the shortest strings, and we will do this by using the rewrite rules to reduce strings to their shortest form.

As an example we prove that $ababab = ba$. Now, by applying the rule $aa \Rightarrow \epsilon$ backwards we get that $ababab = abababaa$, and by then applying $bb \Rightarrow \epsilon$ backwards, we get that $abababaa = ababababba$. We can now apply $abababab \Rightarrow \epsilon$ to get ba .

The above illustrates a common problem found with rule rewriting systems: strings often have to get longer before they can get shorter again. It is also in general impossible to say how much longer a string will get, or which rules to use to rewrite it with. This is the Word Problem in one of its forms.

What we would like is a rule rewriting system for which this is not a problem: one where the rules need only ever be applied left-to-right. Such a system is a *complete* rule system – see later chapters for a formal definition.

It happens that this group has a finite complete rewrite rule system. It is: $aa \Rightarrow \epsilon$, $baba \Rightarrow abab$, $bb \Rightarrow \epsilon$, $cabcbaca \Rightarrow acabcbac$, $cabc bc \Rightarrow acabc b$, $cac \Rightarrow aca$, $cbc abab \Rightarrow bcb caba$, $cbcb \Rightarrow bc bc$, $cc \Rightarrow \epsilon$, $da \Rightarrow ad$, $dbadbab \Rightarrow bdbadb a$, $dbdb \Rightarrow bdb d$, $dcadca \Rightarrow cdcadc$, $dcdbcbaca \Rightarrow cdcdbcbac$, $dcdbcb c \Rightarrow cdcdbcb$, $dcdc \Rightarrow cdcd$, $dd \Rightarrow \epsilon$.

This is very rare, but when it happens it allows us to construct the word acceptor for the group very easily. Such a system, should it exist, can be found by using the Knuth–Bendix algorithm as described in Chapter 6.

Constructing the Word Acceptor

A string will be a least string in this group if it cannot be reduced to a lesser one by any of the rewrite rules. Since the rule system is complete, rules need only ever be applied left-to-right. A string is thus irreducible precisely when it does not contain a left-hand side of one of these rules.

We shall construct an automaton that recognises precisely the reducible strings, by using regular expressions as described in Chapter 1. Let $l \Rightarrow r$ be a rewrite rule; the strings that are reducible by it are precisely those of the form $u \cdot l \cdot v$.

If we adopt the convention that the regular expression $?$ stands for any single letter in the alphabet we are using (in our case $? = (a|b|c|d)$), then this automaton corresponds to the regular expression

$$R_l = (?)^* \cdot l \cdot (?)^*$$

There will be an R_l for each rule $l \Rightarrow r$. Since a string is reducible precisely when it contains a left-hand side, the reducible strings will be those that match

at least one of these regular expressions. We can thus construct a ‘word rejector’: an automaton that accepts precisely the reducible strings. It will correspond to the regular expression

$$R_G = R_{l_1} | R_{l_2} | \cdots | R_{l_k}$$

where l_1, l_2, \dots, l_k are the left-hand sides of the rules. It will look (before determination) like this:

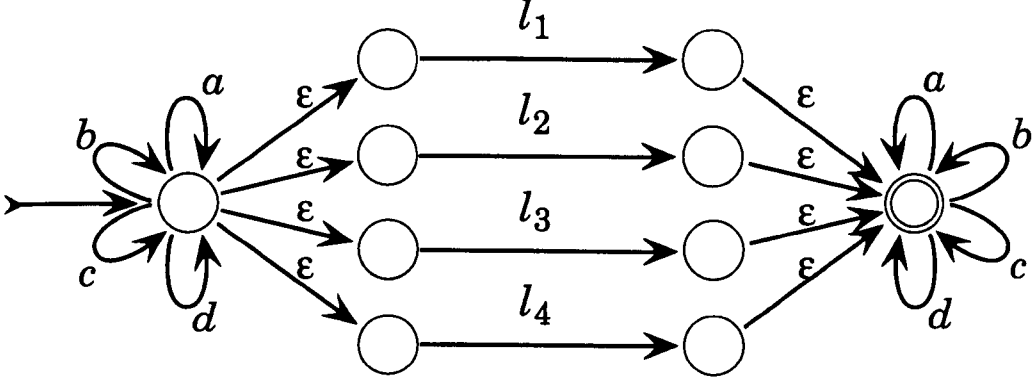


Diagram 3.1: Group word rejector ($k = 4$)

and its complement will be the word acceptor for the group. In this diagram, as in others, the accept states are marked with concentric circles, and the start state, on the left, is marked by a feathered arrow.

Subgroups

The material in the previous sections is all well known; we want to extend it to cover coset systems, and to do this we need to extend the concept of a rewrite rule to cover subgroups.

A different concept of rewrite rule is required because rewrite rules apply globally throughout strings, but subgroup rules are only valid at the start of strings. As an example, in our coset system a , b and c are subgroup generators, so clearly any strings beginning with them can be rewritten as something shorter. If we add the rewrite rules $a \Rightarrow \epsilon$, $b \Rightarrow \epsilon$ and $c \Rightarrow \epsilon$, though, we get our language consisting only of d and ϵ , and the subgroup H is very definitely not of index 2!

We need a new type of rewrite rule: an *initial* rewrite rule is one of the form $Hl \Rightarrow Hr$, and means that $Hl = Hr$ as cosets, and thus that any string lu in the coset system can be rewritten as ru without changing the coset it describes. The previous type of rewrite rule will be called a *global* rewrite rule.

We have a similar concept of completeness for initial rewrite rules, and we are again fortunate in this case that, taken with the previous complete set of

global rewrite rules, the rules $Ha \Rightarrow H$, $Hb \Rightarrow H$ and $Hc \Rightarrow H$ define another finite complete rule set.

Again we can construct a regular expression that describes the strings reducible by the initial rule $Hl \Rightarrow Hr$:

$$R^l = l \cdot (?)^*$$

and we can make a regular expression that matches all initial rules as

$$R^I = (R^{l^1} | R^{l^2} | \dots | R^{l^{k'}})$$

where the initial rules are $Hl^1 \Rightarrow Hr^1$, $Hl^2 \Rightarrow Hr^2$, \dots , $Hl^{k'} \Rightarrow Hr^{k'}$.

We can thus make a word rejector for the coset system: it will accept

$$R_G^I = (R_G | R^I)$$

and will look like this:

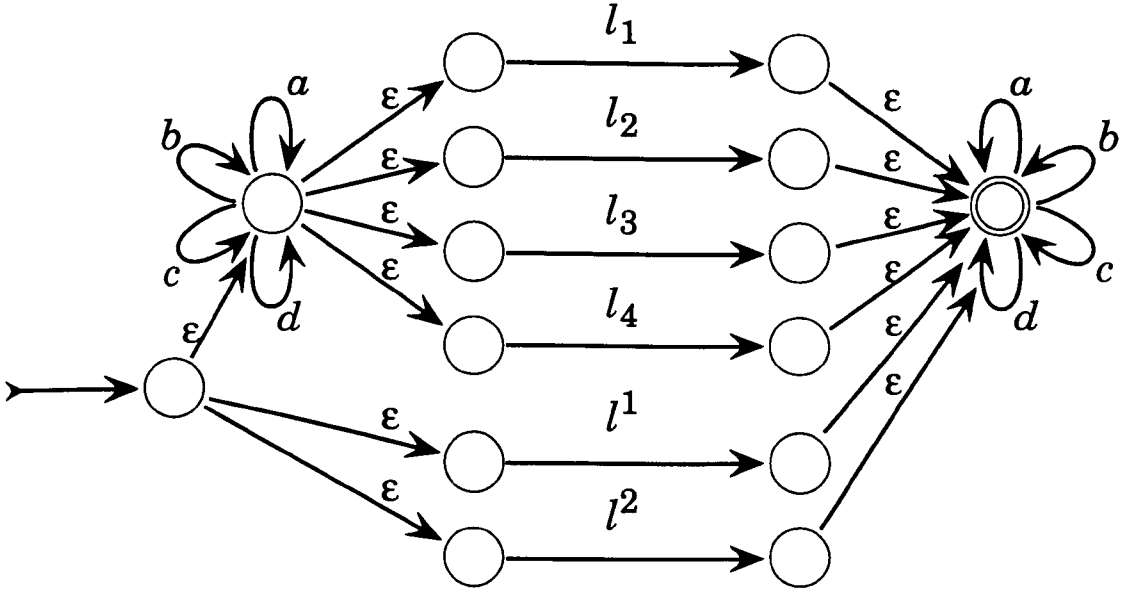


Diagram 3.2: Coset word rejector ($k = 4, k' = 2$)

and its complement will be the word acceptor we are looking for.

It is also possible to make the multipliers from the complete rule set. This is unnecessary for this application, but can be done by applying the algorithm in Chapter 8, which will then terminate after it has processed all the rules. We can thus construct the entire coset automatic system.

Example

The example, G , given at the start of the chapter corresponds to the following situation:

A conjugation matrix is a matrix

$$M = \begin{pmatrix} \alpha & \beta \\ \gamma & \delta \end{pmatrix}$$

where $\alpha, \beta, \gamma, \delta \in \mathbb{C}$, which acts on $z \in \mathbb{C}$ by

$$z \mapsto \frac{\alpha \bar{z} + \beta}{\gamma \bar{z} + \delta}.$$

G can be represented by the following conjugation matrices, approximately determined using a program written by Oliver Goodman:

$$a = \begin{pmatrix} 0 & 1.5672969245 \\ 0.6380411933 & 0 \end{pmatrix}$$

$$b = \begin{pmatrix} -0.99051834 & 0.021117593 \\ 0.8937296094 & 0.99051834 \end{pmatrix}$$

$$c = \begin{pmatrix} 0.6570046777 - 0.8170294774i & -0.142506417 \\ 0.6960550661 & -0.6570046777 - 0.8170294774i \end{pmatrix}$$

$$d = \begin{pmatrix} 0.4550724315 + 0.9641029864i & -0.5792333486 \\ 0.2358039062 & -0.4550724315 + 0.9641029864i \end{pmatrix}$$

Recall that G has a presentation of the form given at the beginning of this chapter; i.e.

$$G = \langle a, b, c, d \mid a^2, b^2, c^2, d^2, (ab)^4, (ac)^3, (ad)^2, (bc)^4, (bd)^4, (cd)^4 \rangle$$

Let C be the circle of centre $-1.0975351456 - 2.0161336273i$ and radius 1.6771876905 , and let $H = \text{Stab}_G(C)$. Then H is also of the form given previously, i.e.

$$H = \langle a, b, c \rangle.$$

We want to enumerate all images of C under G , and for each one we want to know its depth, that is, the length of the shortest word that takes C to it.

Using the above procedure we can make a word acceptor for G/H ; it has 28 states when minimised and is described by the following table. Here, states are listed in Short-Lex order by the first string to reach them, with the start state as state 1. Accept states are marked with a letter 'A', and fail states with a letter 'N'. Note that, because the language is prefix-closed (i.e. any prefix of an accepted string is accepted), there is only one fail state and it is terminal.

	a	b	c	d		a	b	c	d		a	b	c	d			
1	A	2	2	2	3;	2	N	2	2	2	2;	3	A	2	4	5	2;
4	A	6	2	7	8;	5	A	9	10	2	11;	6	A	2	12	7	13;
7	A	14	10	2	3;	8	A	2	2	5	2;	9	A	2	15	2	16;
10	A	17	2	18	3;	11	A	2	19	2	2;	12	A	2	2	7	3;
13	A	2	20	5	2;	14	A	2	15	2	3;	15	A	17	2	21	3;
16	A	2	4	22	2;	17	A	2	12	7	3;	18	A	23	2	2	3;
19	A	6	2	21	8;	20	A	24	2	7	8;	21	A	14	25	2	3;
22	A	2	10	2	11;	23	A	2	26	2	3;	24	A	2	2	7	13;
25	A	27	2	2	3;	26	A	12	2	21	3;	27	A	2	12	28	3;
28	A	2	10	2	3;												

By using depth-first search on this word acceptor (see Chapter 4) we can produce the picture given at the start of Appendix 1, which takes a few hours on one of the University of Warwick's Sun SparcServer 2000's. The circles are coloured by depth, with the nearest circles red (including C , which is the large red central circle) and the furthest circles violet. Formally, saturation and brightness are always 1 and hue is $depth/max_depth$.

Limitations

This method will only work when the coset system involved has a finite complete rule set. This is, in general, unlikely. Even if the group involved has such a rule set, there is no guarantee that the coset system will.

As an example, the other half of the circle packing given by the above group is produced by taking the orbit under G of the circle which has centre $0.1607775601 + 0.1838178602i$ and radius 0.6264092653. This circle is stabilised by the subgroup H' generated by b , c and d .

There is no finite complete rewrite rule set for this system. It is, however, still possible to find its word acceptor, and this will be described in Chapters 6 and 7. It has 47 states and was used to draw the second picture in Appendix 1, which incorporates both halves of the packing.

There is one case in which a finite complete rule set can be guaranteed, and that is when G is a free group. As long as the generators in A freely generate G (i.e. we don't have any surplus generators) and H is a finitely generated subgroup, G/H will have a finite complete rewrite rule set. This is discussed further in Chapter 5.

Chapter 4: Coset Enumeration by Depth-first Search

Aim

This section describes the use of coset automata to do coset enumeration; that is, to list representatives of all cosets of a subgroup within a group, and to do so efficiently. An algorithm based on depth-first search is given, and some wild claims are made about its speed and storage use.

This section can be skipped by those with a Computer Science background.

Example Case

Consider the following Short-Lex word acceptor for the subgroup $H = \langle s, tst^{-1} \rangle$ of the free group $G = \langle s, t \rangle$ with $S = s^{-1}$, $T = t^{-1}$:

	s	S	t	T		s	S	t	T		s	S	t	T
1 A	2	2	3	4;	2 N	2	2	2	2;	3 A	2	2	5	2;
4 A	6	7	2	4;	5 A	6	7	5	2;	6 A	6	2	5	4;
7 A	2	7	5	4;										

It looks like this:

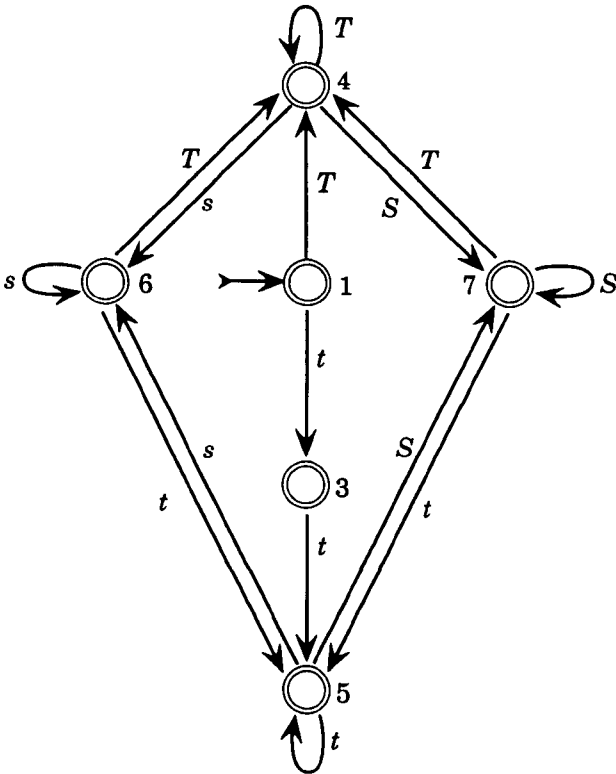


Diagram 4.1: Enumeration example

We want to enumerate all the cosets of H in G . Since there are infinitely many of these, we have to decide when to stop, and which subset of the cosets

we actually want to enumerate. Usually we will have some object associated to each coset – the translation of a starting circle, for example. We may be uninterested in circles that are very small, or a long way from the origin, so we could decide to ignore all further translations of such uninteresting circles, and go and count more interesting ones instead.

We can formalise this as follows: We have a pruning function p on our language, which takes values in $\{true, false\}$, and if, for some word w , $p(w) = true$, then for all words of the form $w \cdot u$, we have $p(w \cdot u) = true$. Our intention is to enumerate all words w for which $p(w) = false$.

There are two sensible approaches to this problem, depending on how large the set of words will be: depth-first and breadth-first search.

If the number of words is small enough to be produced quickly, then it doesn't matter which way we produce them. If they are going to take a long time, then in general we would probably like them in length order, because the shorter words generally correspond to less complex objects in the enumeration we are constructing – e.g. larger or nearer circles.

To produce a list in length order requires a breadth-first search: one which produces all the words of length n before moving on to length $n + 1$. The most sensible way to do this is to store all the words of length n , because otherwise we shall have to make each word many times while examining its children at each length, which is a lot of wasted work.

Breadth-first search to length 3 in the above automaton produces the following sequence of words:

Length 0: ϵ

Length 1: t, T

Length 2: tt, Ts, TS, TT

Length 3: $tts, ttS, ttt, Tss, Tst, TsT, TSS, TSt, TST, TTs, TTS, TTT$.

This method is simple, lists the words in Short-Lex order so it can easily be terminated once enough words have been found – and quickly runs out of memory at anything bigger than about length 10. Since we may well want to enumerate as far as length 1000, this is not a good approach. In general, coset automata will have exponential growth of number of words against length, and so breadth-first enumeration is always going to be a bad idea: either an exponential amount of storage is required, or an exponential amount of time is spent retracing routes already taken.

The great advantage of automata over more traditional methods is that an

alternative is possible: depth-first search. In this method we begin at the start state with the empty word, and at each state we reach we take the first arrow out, until we have reached a word w with $p(w) = \text{true}$. We then retrace our steps along the last arrow taken, and take the next arrow (if there are any left) instead and keep going. If there are no unused arrows from a state, we again backtrack. We keep this up until we are back at the start state again having taken all arrows out, and then we stop.

As an example, suppose we were depth-first searching the above automaton to length 3. We would begin at state 1 with the empty word ϵ . The first arrow available is a t , taking us to state 3. From here we can only take a t , so we get to state 5. From state 5, we can take an s , and we have now reached length 3. We now replace the final s by a S , and then by a t . This gives us all the strings starting with t , so we backtrack all the way to the start, and begin again with the strings starting T . This takes us to state 4, where we have a choice of arrows, so we take the first one, s , to state 6.

Continuing the procedure, we end up with the following list of words:

$\epsilon, t, tt, tts, ttS, ttt,$

$T, Ts, Tss, Tst, TsT, TS, TSS, TSt, TST, TT, TTs, TTS, TTT$

Note that the words come out in lexicographic (or dictionary) order, and in no relation to their length.

Advantages and Disadvantages

There are two disadvantages to this method: the pruning method must be chosen at the start of the enumeration (it's not possible to let it run a little longer and get a few more words) and the answers come out in the wrong order (dictionary as opposed to Short-Lex). The advantages, however, are huge from a computational point of view: the storage improvement is dramatic. The storage used is linear in the maximum allowed length (I normally use about 5000), while for breadth-first it is typically of the order of $(k - 1)^n$ for a search to length n of a k -generator group. The time required for both methods is the same, since they enumerate the same number of words.

Depth-first search is pretty much optimal for the kind of work I use coset enumeration for: typically group elements are 2×2 complex matrices acting either on points or on circles (as the image of three points). For such examples depth-first search imposes no storage overheads, only one matrix multiplication is done per group element generated, and almost all the program time is spent doing the floating-point work required to multiply the matrices, produce the image of the circle or point, and decide on pruning.

Chapter 5: An example – Punctured Tori

Aim

This section describes the construction of an automatic coset system, which is used to solve a specific problem of enumerating cosets in free groups.

The first part is devoted to a description of the problem to be solved: drawing the limit sets of a particular family of Kleinian groups. It is followed by an explanation of the use of a coset automatic system (specifically a word acceptor) to give a solution.

This work is joint work with John Parker and Greg McShane of the University of Warwick and is described in [MPR].

The Problem

It is possible to embed the Teichmüller space of the punctured torus as a holomorphic family of Kleinian groups $\{G_\mu\}$ using the Maskit Embedding – for further details see [KS], [W], [Mas].

I shall not discuss the geometry here; instead I shall describe the problem of drawing the limit sets of these Kleinian groups.

$\mathbf{T}_{1,1}$ (the Teichmüller space of the punctured torus) is represented as a space of discrete subgroups of $\mathbf{PSL}_2(\mathbb{C})$, the group of projective complex 2×2 matrices of the form

$$\begin{pmatrix} a & b \\ c & d \end{pmatrix}$$

acting as the linear fractional transformation

$$z \mapsto \frac{az + b}{cz + d}.$$

A discrete subgroup G of $\mathbf{PSL}_2(\mathbb{C})$ is called a *Kleinian group*. The subset $\Omega = \Omega(G)$ of \mathbb{C} on which G acts properly discontinuously is called the *regular set* and its complement is called the *limit set* of G .

The Maskit Embedding of $\mathbf{T}_{1,1}$ produces groups parameterised by the complex number μ in the following way:

Let s be the element $z \mapsto z + 2$ and let t_μ be the element $z \mapsto \mu + \frac{1}{z}$. Then $G_\mu = \langle s, t_\mu \rangle$.

In [KS] the following method of parameterising elements of G_μ on the boundary of the embedding is described: For the μ values we are interested in,

G_μ is free on two generators and Kleinian. An element of G_μ is *parabolic* if its corresponding matrix has trace ± 2 ; it is *generically parabolic* if its matrix has trace ± 2 for all possible values of μ , otherwise it is *accidentally parabolic*. The element s is clearly always parabolic; the only generically parabolic elements will be conjugates of powers of s and $st_\mu st_\mu^{-1}$.

We choose μ to force a particular element $W_{p/q}$ of G_μ to be parabolic; this involves finding its trace as a polynomial in μ and finding a value of μ to set that to ± 2 . This is called *pinching* along $W_{p/q}$.

The word $W_{p/q}$ is chosen so that the group G_μ will be on the boundary of the embedding. Each group on this boundary corresponds to a particular μ and a coprime pair of non-negative integers p and q with $p \leq q$ for which $W_{p/q}$ has trace ± 2 .

To find $W_{p/q}$, [KS] enumerates the rationals by Farey sequences. Two rationals p/q and p'/q' are called *Farey neighbours* if $pq' - qp' = \pm 1$. For a pair of Farey neighbours we define the *Farey sum* as

$$\frac{p + p'}{q + q'},$$

which is a neighbour of both p/q and p'/q' , which are called its Farey parents. All positive rationals are obtained uniquely by repeated application of this process, starting with $0/1$ and $1/0$.

We define the words $W_{p/q}$ inductively via

$$W_{1/n} = s^{-1}t^n$$

$$W_{n/1} = s^{-n}t$$

and, if p/q and p'/q' are Farey neighbours with $p/q < p'/q'$, then

$$W_{(p+p')/(q+q')} = W_{p'/q'} \cdot W_{p/q}.$$

If p/q and p'/q' are Farey neighbours then [KS] shows that $W_{p/q}$ and $W_{p'/q'}$ generate the free group G_μ .

We shall thus, for a given p/q , find a corresponding μ and draw the limit set of the group G_μ . The parameter μ will be assumed from now on.

The Problem in Coset Terms

The limit set of G in this embedding is the closure of the orbit of the real axis. Each image of \mathbb{R} under G is a circle or a line, and to draw the limit set we can equivalently draw all these circles and lines.

Each image of \mathbf{R} corresponds to a coset of $Stab_G(\mathbf{R})$, the stabiliser of the real line, which we shall call H .

H is generated by s and $t^{-1} \circ s \circ t$. For historical reasons, matrices multiply on the left but coset systems multiply on the right; bearing in mind the evident opportunities for confusion, we shall enumerate the coset system G/H where $G = \langle s, t \rangle$ and $H = \langle s, tst^{-1} \rangle$, as described in Chapter 4.

Free Groups are Always Coset Automatic

This coset system is coset automatic. We shall prove this by showing that it has a finite complete rewrite rule system as was described in Chapter 3. This proof is valid for any finitely-generated subgroup of a finitely-generated free group with any set of inverse-closed free generators. It is a special case of the general existence theorem in Chapter 10, because free groups are 0-hyperbolic.

Any finitely-generated subgroup $H = \langle h_1, \dots, h_m \rangle$ of a finitely-generated free group $G = \langle g_1, \dots, g_n \rangle$ is necessarily ε -*quasiconvex*, that is, if v is a shortest string in G to an element of H , then $\overline{v(t)}$ is never further than some constant ε from H .

To show this, let u be a shortest string in the generators of H with $\bar{u} = \bar{v}$. We consider u as a string in the generators of G , which can then be reduced to v . Because G has a finite complete rewrite rule set E_G (consisting of rules of the forms $Bb \Rightarrow \epsilon$ and $bB \Rightarrow \epsilon$), we can make u from v by repeatedly deleting strings of the form bB or Bb , starting from the front each time. Any letter, say the t th, that survives into v must come from an identifiable letter, the t' th, say, in u , and, because we are only removing substrings equal to the identity, $\overline{u(t')} = \overline{v(t)}$.

Any point on u is at most half the length of a generator of H from a point in H , and so, by the above, is any point on v . Hence H is ε -quasiconvex in G with

$$\varepsilon = \max \left\{ \left\lfloor \frac{|h_i|}{2} \right\rfloor \right\}.$$

Let E_H be E_G together with the set of all rewrite rules of the form $Hx \Rightarrow Hy$ with $|x| \leq \varepsilon + 1$. We shall show that any reducible string can be reduced using E_H , and thus that E_H is complete, since only irreducibles can not be reduced by it.

Suppose $Hu = Hv$ with $v < u$ but with both u and v reduced with respect to E_G . Set $u = pr$ and $v = qr$ where r is the largest (possibly trivial) common suffix of u and v . We must now have $\bar{p} \cdot \bar{q}^{-1} \in H$, so that $Hp \Rightarrow Hq$ is a rewrite

rule. If u is to be a counter-example to our claim of completeness, we must have $|p| > \varepsilon + 1$, or else $Hp \Rightarrow Hq$ is in E_H .

$\bar{p} \cdot \bar{q}^{-1}$ is a geodesic to a point in H , because G is free and we have removed any common suffix, so it always lies within ε of H ; thus $\overline{p(\varepsilon + 1)}$ lies within ε of some $h \in H$, and $h \neq 1$ since $d(\overline{p(\varepsilon + 1)}, 1) = \varepsilon + 1$. So there is some y from H to $p(\varepsilon + 1)$ with $\overline{p(\varepsilon + 1)} = h\bar{y}$ and $|y| \leq \varepsilon$.

We can thus deduce the rule $Hp(\varepsilon + 1) \Rightarrow Hy$, and since $u = pr$, this rule applies to reduce u . So Hu is reducible under a rule that must be in E_H .

We can easily create E_H – in fact, for this particular case it is enough to cut the generators of H in half and use those. In practice, I actually use a Knuth–Bendix algorithm as described in Chapter 6, which will produce the complete rewrite rule set in a few milliseconds. We can then apply the techniques in Chapter 3 to make the word acceptor. Note that, because of Chapter 2, the condition that the generating set be inverse-closed and free is unnecessary to guarantee being coset automatic; it is needed for finite completeness, however.

Enumerating the Circles

To each word w accepted by the word acceptor there corresponds a unique circle (or line) that is an image of the real line, and a matrix that is the transformation taking \mathbf{R} there.

We wish to plot all such images of \mathbf{R} , but are not interested in any circles which are too small or are too far from the origin to be visible on our page. Typically we will only want to see circles at least partly contained in the strip $-2 \leq \mathbf{R}(z) \leq +2$ and with radius at least 10^{-5} , which corresponds to diameter one dot on a 300 dots per inch printer (this is the *visibility radius*).

We use the depth-first search algorithm from Chapter 4, but we can now use a better pruning method. Because t ($= t_\mu$) contains an inversion, it is never sensible to eliminate a circle simply because it is a long way out. Since there are only finitely many circles within the area we are looking at, however, there *will* be a minimum radius that we can prune at and be sure that we are not missing any circles large enough to be visible. Typically 10^{-7} works as a minimum radius, giving us all circles in the strip we want to see.

Depth-first search is pretty much the optimal method for this type of solution – only one matrix multiplication is needed per circle generated, there are no storage overheads and nearly all of the program time is spent doing the floating point work required to multiply the matrices and determine three points in the image of \mathbf{R} , and from them the circle's centre and radius.

Changing Generators

A large improvement to the performance of the algorithm as stated here can be made by changing the generators, and was suggested by Greg McShane. The generator s merely duplicates the picture along the real axis, and if w produces a visible circle, ws^n and wS^n are still the same size, but well off the piece of paper. The program will thus spend most of its time looking at and generating circles that are quite irrelevant.

The solution is to use a better pair of generators, which will still generate G , but with respect to which s is a long word. One of the generators chosen is the word $w = W_{p/q}$ used to obtain μ ; the other is $u = W_{p'/q'}$ where p'/q' is the first Farey parent of p/q . Since w is of length $p + q$ in s and t , s is of a similar length in w , and so duplication along the real axis is much less frequent.

In fact, w has a fixed point on the circle tangent to the real axis at -1 , so has the effect of drawing the picture in towards that point, so that words close to it are generated first, leading to even better results.

Unfortunately, for large w the drawing-in effect is so marked that large parts of the visible area are left completely blank, since s is now a very long word. It is best, then to use a w of length no more than 20, but that corresponds closely to the p/q used to generate the picture; we generally use an early term in the continued fraction expansion of p/q .

Performance

Since G is free, G/H has a finite complete rule set for any pair of generators. The time to generate the word acceptor is therefore insignificant – maybe a tenth of a second. Since we are using depth-first search, memory usage is also small – maybe 350Kb.

Using the program on a desktop Sun IPC, a useful plot (e.g. to depth 12) can be generated in a minute, and higher quality plots (going as far as, perhaps, depth 250) in about an hour, corresponding to several million circles being examined and several tens of thousand being plotted. On one of the University of Warwick's fast SparcServer 2000's, this time is reduced to a matter of minutes.

Hausdorff Dimension

This method has been used by Greg McShane to estimate the Hausdorff dimension of the limit set.

We do this by plotting $-\log_{10}(\text{radius})$ along the x -axis with a scale of, say, 100 samples per unit, and plotting the number of circles of radius between 10^{-x} and $10^{-(x+0.01)}$ on the y -axis. The gradient of this line is approximately the Hausdorff dimension of the limit set.

An even more accurate result can be obtained by plotting the log of the number of circles on the y -axis instead, and that is the picture shown here. This radii distribution is for $p/q = 5/16$ with a minimum radius of 10^{-7} .

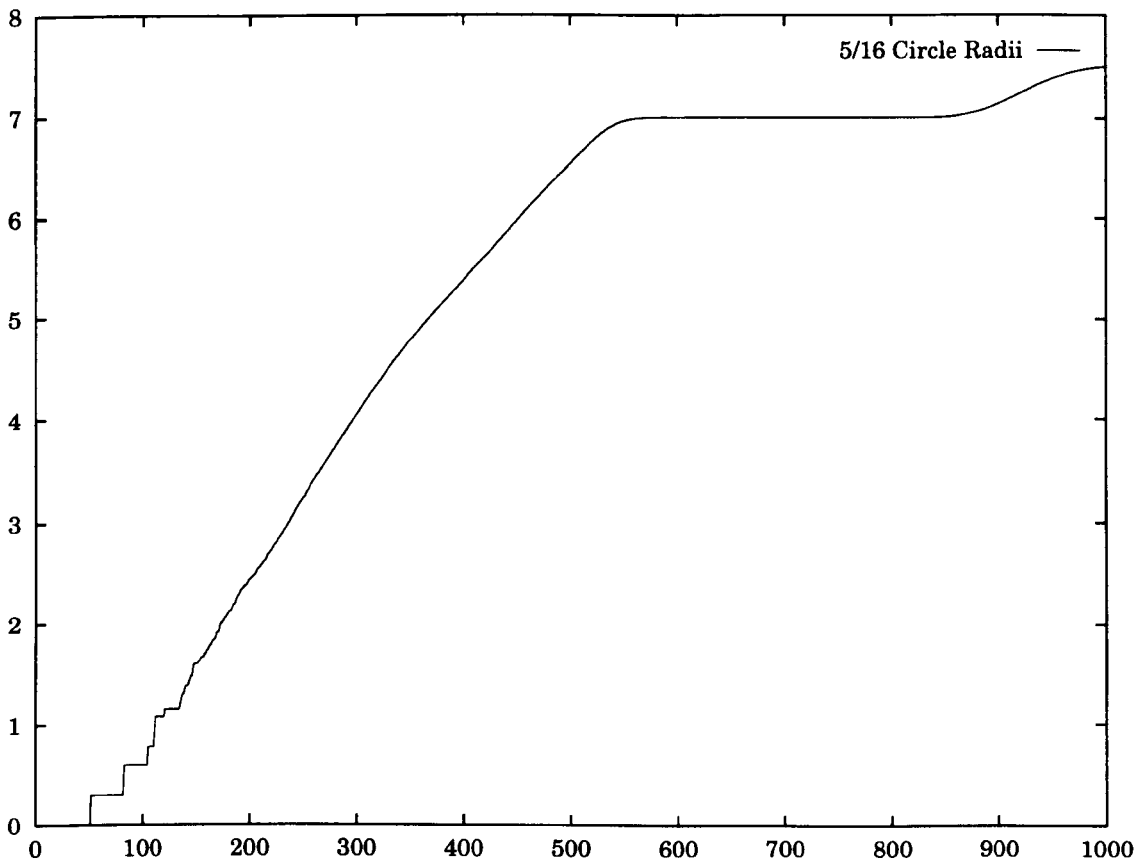


Diagram 5.1: Hausdorff dimension

To construct the graph we start with 1000 empty buckets. For each circle of radius r generated, we increment the count of bucket $\lfloor -100 \log_{10}(r) \rfloor$. When the run is completed, we plot on the y -axis $\log_{10}(\text{count in bucket } x)$.

For further details of this construction (which is actually the circle packing exponent) and John Parker’s proof that it is equal to the Hausdorff dimension, see [MPR].

Chapter 6: Knuth–Bendix

Aim

Given a group presentation and some subgroup generators, we want to work out how to multiply and reduce words so that we can produce a word acceptor and multipliers.

We do this using rewrite rules of the form $left \Rightarrow right$, in which the two sides of the rule are equal, but the right-hand one is shorter, and thus a string can be rewritten using this rule as a shorter equivalent string.

We need some method for producing rewrite rules from the group presentation – in particular, we need a method that will guarantee we get all necessary rules. This chapter is a description of one such method, the Knuth–Bendix algorithm, which is ideal for this problem. This method is also used in the older word-difference algorithm for making automatic groups – see [CEHLPT].

First, we need some definitions.

Monoid Presentations

Let S be a monoid (a semigroup with an identity) and $R \subseteq S \times S$. The *quotient* S/R of S by R is the set of equivalence classes of S under the equivalence relation \sim generated by the equivalences $aub \sim avb$ for all $a, b, u, v \in S$ with $(u, v) \in R$.

S/R has a monoid structure inherited from S , and the quotient map $q: S \rightarrow S/R$ is a monoid homomorphism with the property that $q(u) = q(v)$ for any $(u, v) \in R$. Moreover, it is the universal such map, in the sense that any other map from S with this property must factor through it.

Recall that A^* is the set of all strings over an alphabet A . We can consider A^* to be a monoid with multiplication being concatenation; in this case the empty word ϵ is the identity.

If $R \subseteq A \times A$ and $Q = A^*/R$ we say A and R form a presentation for Q and $Q = \langle A \mid R \rangle$.

The pairs $(u, v) \in R$ are called *relations* for Q and the image $q(u)$ of a string $u \in A^*$ in Q is generally written \bar{u} .

A group presentation $\langle g_1, \dots, g_n \mid r_1, \dots, r_m \rangle$ can be considered to be a monoid presentation by adding the formal inverses G_1, \dots, G_n to the alphabet, together with rules of the form $(g_i G_i, \epsilon)$ and $(G_i g_i, \epsilon)$ linking them together, and

turning relators r_i into pairs (r_i, ϵ) . The above presentation would then be

$$\langle g_1, G_1, \dots, g_n, G_n \mid (r_1, \epsilon), \dots, (r_m, \epsilon), (g_1 G_1, \epsilon), (G_1 g_1, \epsilon), \dots, (G_n g_n, \epsilon) \rangle.$$

Conversely, $\langle A \mid R \rangle$ is a group if for each $g \in A$ there is some $G \in A^*$ such that $\overline{gG} = \overline{Gg} = \epsilon$.

Rewrite Rules

For an ordered alphabet A , a *rewrite rule* or *rule* over A is a formula of the form $l \Rightarrow r$ where $l, r \in A^*$ and $l > r$ in the Short-Lex order. l and r are the *left-hand* and *right-hand* sides of the rule respectively.

We can apply these rules to strings to turn them into other lesser strings. Let $u, v \in A^*$ and let E be a set of rules. We say $u \rightarrow v$ if there are $a, b \in A^*$ and a rule $l \Rightarrow r$ in E such that $u = alb$ and $v = arb$. We say that v can be obtained from u by applying the rule $l \Rightarrow r$. Note that necessarily $v < u$ in Short-Lex order.

Let \rightarrow^+ be the transitive closure of \rightarrow , i.e. $u \rightarrow^+ v$ if there is a finite chain of reductions

$$u \rightarrow u_1 \rightarrow u_2 \rightarrow \dots \rightarrow v$$

We say u *reduces* to v and v is a *reduction* of u .

Let \rightarrow^* be the reflexive closure of \rightarrow^+ (i.e. $u \rightarrow^* u$), and \leftrightarrow^* the symmetric closure of \rightarrow^* (i.e. $u \leftrightarrow^* v \iff v \leftrightarrow^* u$). Then \leftrightarrow^* is exactly the definition of \sim , the equivalence relation on monoids given earlier, and the set of \leftrightarrow^* equivalence classes can be identified with the monoid $\langle A \mid E \rangle$ where E is considered to be a set of pairs (l, r) .

Since Short-Lex is a well-ordering, all chains of reduction eventually terminate, and so any string u is reducible to some *irreducible* string v which cannot be further reduced.

Completeness

For a general set of rules the irreducibles need not be unique, in that there may be strings $u \neq v$ with $\bar{u} = \bar{v}$ but neither u nor v containing the left-hand side of a rule.

A set of rules E is *complete* if this is not the case, i.e. for all $u \in A^*$, there is a unique irreducible $v \in A^*$ with $u \leftrightarrow^* v$. An equivalent condition is that if

$u \rightarrow v_1$, and $u \rightarrow v_2$ then there is a v with $v_1 \rightarrow^* v$ and $v_2 \rightarrow^* v$ – this is called *confluence* of the rules. See [CEHLPT], [EHR] or [KB] for further details.

Finite complete rule sets are valuable because they allow fast reduction of strings to their least equivalent form. For any u , there is a finite sequence of applications of rules that will take u to v , an irreducible, and v is unique. But, since u is of finite length, there are only finitely many applications of rules possible. So if we keep applying any rules that will work in any order we like will eventually reach an irreducible – and, by completeness, this must be v . So the order of application of rules is irrelevant.

We can now easily reduce strings – just read along the string until we have read a left-hand side of a rule, replace that by the right-hand side, and start again. Similarly we can easily identify the irreducibles: they are the strings that contain none of the left-hand sides in the rule set, which is something easily described by a finite state automaton.

We need some way to do the following: tell when a rule system is complete; work out what rules need to be added to a rule system to make it complete; and cope with rule systems that have no finite completion.

Knuth–Bendix

The Knuth–Bendix algorithm described in [KB] provides us with a method for checking if a rule system is complete and for determining which rules need to be added to make it complete.

We can test for confluence (or completeness) as follows: Let E be a set of rules over A . Then E is complete if and only if for all $a, b, u, v, y \in A^*$ with $y \neq \epsilon$

if $ay \Rightarrow u$ and $yb \Rightarrow v$ are rules in E , then for some $t \in A^*$ there are reductions $ayb \rightarrow ub \rightarrow^* t$ and $ayb \rightarrow av \rightarrow^* t$.

if $y \Rightarrow u$ and $ayb \Rightarrow v$ are in E then for some $t \in A^*$ there are reductions $ayb \rightarrow aub \rightarrow^* t$ and $ayb \rightarrow v \rightarrow^* t$.

The above conditions are clearly necessary for completion. We can prove sufficiency by induction as in [EHR] or [CEHLPT] Section 6.

Suppose all strings less than w are reducible to a unique irreducible and consider two possible reductions of w to irreducibles using rules $u_1 \Rightarrow v_1$ and $u_2 \Rightarrow v_2$ from E as the first reductions.

If u_1 and u_2 do not overlap in w , then $w = au_1bu_2c$ for some $a, b, c \in A^*$. So, applying the two rules we have $w \rightarrow av_1bu_2c$ and $w \rightarrow au_1bv_2c$. Both of

these reduce to av_1bv_2c , which reduces to some unique irreducible s , since it is less than w . Since av_1bu_2c and au_1bv_2c are less than w , and both reduce to s via av_1bv_2c , they reduce uniquely to s . So, since the two given reductions of w pass through strings that can only reduce to s , both the irreducibles that w reduces to must be s .

If the left-hand sides u_1 and u_2 overlap in w , then they do so in one of the two ways described above, and we can apply the same technique. w must be of the form $caybd$, and both one-step reductions will give strings that are smaller than w , and these, by induction, must each reduce to a unique irreducible that can be obtained by any route. But, again, both strings are reducible to ctd , so the two irreducibles they reduce to must be the same. So both reductions of w give the same answer.

We can now apply induction and show that any string can be reduced to a unique equivalent irreducible, and so E is complete.

We can use the above to construct an algorithm (the Knuth–Bendix algorithm) for testing E for completeness. If we test each pair of rules in E for overlaps (including overlaps of a rule with itself), and reduce the corresponding two strings as above, then we must reach a common irreducible.

If this algorithm fails for any pair of strings u and v then we get two irreducibles under E which are different reductions of the same string. We must therefore add a rule relating them to E for E to be complete, and so, assuming $u > v$ we add $u \Rightarrow v$ to E .

If we iterate this process, we will either end up with a finite complete set of rules or we will get an ever-increasing set. If we reach a finite set, then we can easily build an automatic system as described in Chapter 3. The more interesting (and more common) case is when there is no finite complete set, and so we have to do something more complicated.

A *k-complete* set of rules is a set that is complete for strings of length at most k , i.e. for all $u \in A^*$ with $|u| \leq k$, there is a unique irreducible $v \in A^*$ with $u \leftrightarrow^* v$.

Since we never throw rules away, if we consider the restrictions of the increasing sets of rules produced by the Knuth–Bendix to left-hand sides of length at most k , these must become constant after a finite time $t(k)$, giving us a *k-complete* set. We shall call this restricted set of rules E_k . This will be the foundation of the work in the next chapter, where we estimate what an automaton really is based on its restriction to words of length at most k .

Estimating k -completion

It is certain that after some time $t(k)$ the set of rules produced by the Knuth–Bendix will be k -complete. We need some way to determine when this has happened, based on the rate of growth of the lengths of the rules produced. This is, of course, theoretically impossible (it is a case of the Word Problem), but that need not stop us developing some heuristics.

Currently I use three methods: gap, threshold and absolute.

Assume that the number of rules produced so far is t , and that the current rule has a left-hand side of length k_t . We keep an array of numbers *last_new* and set $\text{last_new}[k_t] = t$ whenever a rule appears. We also keep an array *max_gap* which is set to the largest value so far taken by $t - \text{last_new}[k_t]$, i.e. the largest number of rules between two occurrences of a rule of left-hand side length k_t .

A length k is considered to be *dubious* if it is unlikely that the rule set is k -complete. Every time a rule appears, we consider all k starting from 0 until we find a dubious one; we then assume that the rule set is $(k - 1)$ -complete.

The gap test for k states that k is dubious if after t rules

$$\text{last_new}[k] > t - \text{gap_factor} \times \text{max_gap}[k]$$

where *gap_factor* is normally 1 but may be set by the user.

The threshold test for k states that k is dubious if after t rules

$$\text{last_new}[k] > (1 - \text{certainty_threshold}) \times t$$

where *certainty_threshold* is normally 0.3 but may be set by the user.

The absolute test for k states that k is dubious if after t rules

$$\text{last_new}[k] > t - \text{ok_words}$$

where *ok_words* is normally 25 but may be set by the user.

Normally all three methods are applied in parallel, and a k that fails any of them is considered to be dubious.

Training

The above default values of the parameters *gap_factor*, *certainty_threshold* and *ok_words* are a sensible compromise for many groups, but it is possible to get much better figures for particular cases, depending on the growth rate of the Knuth–Bendix rules.

I use the following training heuristic: we run the test after each rule is read, and if, after running the test, we have a larger first dubious k than the one we had last time, I modify the parameter associated to the test that failed that k to weaken it, so that next time it is more likely to pass it. This is based on the assumption that this test is being stricter than the other two, so should be weakened to bring them into line.

The standard weakening changes I use are to reduce *gap_factor* by 0.03, to reduce *certainty_threshold* by 0.02 or to reduce *ok_words* by 4, never letting these parameters fall below reasonable limits.

It is, of course, quite possible for this method to make a mistake and for a rule to be generated by the Knuth–Bendix process after its left-hand side length has stopped being considered dubious. If this happens, then the rule that has been weakened is instead strengthened, by 0.08, 0.05 or 10 respectively, to make it less likely it will make that mistake again. We must also recompute any calculations we have made based on any assumptions of having a k -complete set.

Describing a Coset System

We now have to work out how to get the coset system we are interested in into a form suitable for describing with rewrite rules.

Consider the group $G = \langle g_1, \dots, g_n \mid r_1, \dots, r_m \rangle$ and its finitely-generated subgroup $H = \langle h_1, \dots, h_\nu \rangle$. Assume that $A = \{g_i : i = 1, \dots, n\}$ is inverse-closed, and similarly for $B = \{h_i : i = 1, \dots, \nu\}$, where each $h_i \in A^*$. Add a new letter, H , to A to give $A' = A \cup \{H\}$.

There are to be two types of rewrite rules: global rules of the form $r_i \Rightarrow \epsilon$ for each relator, and initial rules of the form $Hh_j \Rightarrow H$ for each generator of H .

The Knuth–Bendix algorithm gives us a sequence of increasing sets of rewrite rules for strings in A'^* , with k -complete sets E_k for each k ; let E_∞ be the union of all these. We would like to work in the coset language HA^* where all strings contain precisely one H , at the start. In fact, this set of rewrite rules is exactly the one we want.

First, since all rules in E_∞ are obtained from initial and global rules using the Knuth–Bendix overlap procedure, we can show that any rules generated can contain at most one H , and if it occurs it must be at the start.

Certainly any rewrite rules produced by the Knuth–Bendix process will be true in the coset system; it remains to show that all irreducibles under E_∞ are truly the Short–Lex least representatives in the coset system.

Suppose $H\bar{u} = H\bar{v}$, with $v < u \in A^*$. Then, for some $\bar{w} \in H$, $\bar{w}\bar{u} = \bar{v}$, and we may assume that w is the concatenation of a series of the semigroup generators B of H .

Let w , u and v be of lengths a , b and c respectively; then $c \leq b$. If we run the Knuth–Bendix algorithm for $t(a + b + 1)$ iterations to get an $(a + b + 1)$ -complete system, we will get a chain of rewrite rules in E_{a+b+1} leading from wu to some irreducible s and another from v to s , since this rewrite is definitely possible in G .

Now, we will get Hs reducing under E_{a+b+1} to some Hs' . As wu and v both reduce to s , both Hwu and Hv can reduce to Hs , so, by $(a + b + 1)$ -completeness, Hwu and Hv reduce to Hs' . Since w is in terms of the generators B of H , we know that Hwu can also reduce to Hu , so we must get Hu and Hv both reducing to Hs' .

Thus for any pair of strings u and v with $H\bar{u} = H\bar{v}$, there will be a chain of rules reducing Hu and Hv to a common Hs' . So if Hu is truly reducible to an irreducible Hv , it will be reducible under this system.

Thus any reduction in the coset system can be done by using the rules.

Improvements

There are many improvements that can be made to this naïve algorithm. It is clearly wasteful to keep rules that have been superseded, that is, ones for which the right-hand side or a proper substring of the left-hand side are reducible, and at any time these may be replaced by their reduced equivalents without affecting the irreducibles under E .

Also, if we are working in a group and generate a new rule $u \Rightarrow v$, then uv^{-1} is a relator in the group, so all cyclic permutations of it and its inverse can also be added. Similarly, if we generate the rule $Hu \Rightarrow Hv$, then uv^{-1} is a subgroup generator, so vu^{-1} is also.

Further, by appropriate application of inverses, it is never necessary for the difference between the lengths of the left and right-hand sides of a rule to be greater than 2.

Finally, much can be done to enable fast lookup of rules for word reduction, which is by far the most common operation in the algorithm. Work on this is being done by Julia Dain of the University of Warwick Computer Science Department.

Chapter 7: Automaton Induction

Aim

After the preceding chapter we are hopefully in a situation where we know all the strings up to a given length k accepted by an automaton. Now, if the strings happened to be $\{\epsilon, a, aa, \dots, a^k\}$, it would be a reasonable guess that the automaton was actually accepting the language $(a)^*$. Even in more complex cases we might be able to find a pattern in the strings and guess what the automaton would be. This is the method depicted in the diagram.

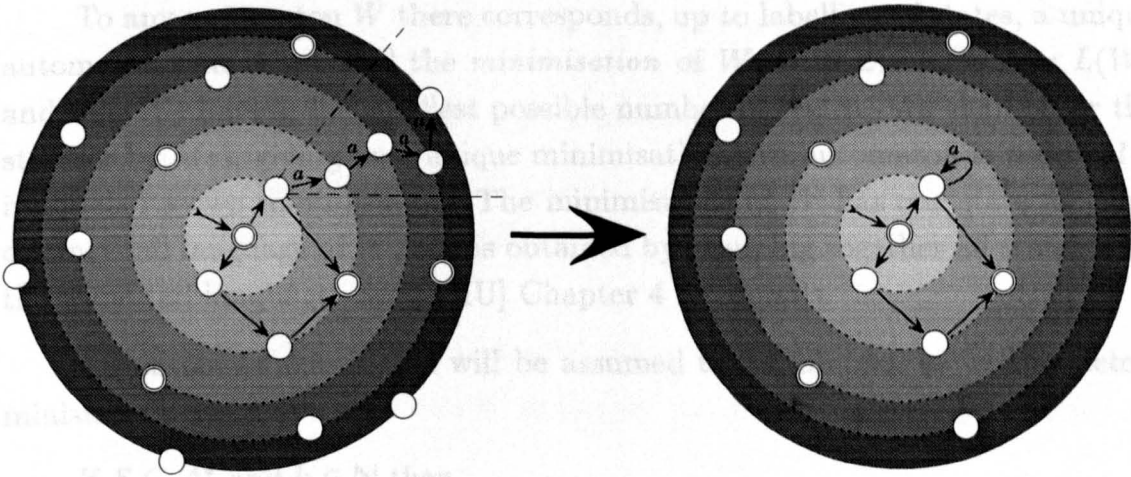


Diagram 7.1: Induction Example

This chapter will describe a method for determining the automaton, given the strings it accepts up to a length k . It will also find bounds on k that ensure that the method will always work, and that it will produce the right answer.

This algorithm is to be found in the work of Trakhtenbrot and Barzdin [TB] who refer to earlier work of Moore [Mo]. Some of the methods used in implementing the algorithm are based on the work on word counting in [SH].

Definitions

Let W be a deterministic finite state automaton over an ordered finite alphabet A . We shall use the Short-Lex ordering on A^* .

If $u \in A^*$ and s is a state of W , then $W[s][u]$ is the state of W reached by following the path labelled by u from s . $W[u]$ is $W[s_0][u]$ where s_0 is the start state of W . States of the form $W[s][a]$ where $a \in A$ are called the *children* of s .

For any state s of W there is a least word $\lambda(s)$ with $W[\lambda(s)] = s$.

If s is a state of W , its depth $\delta(s) = |\lambda(s)|$. The depth of an automaton W , written $\delta(W)$, is the maximum of the depths of its states.

If s is a state of W , its tail language is defined by

$$L(W; s) = \{w \in A^* : w \text{ is an acceptable string from } s \text{ in } W\}.$$

Equivalently,

$$L(W; s) = \{w \in A^* : \lambda(s) \cdot w \in L(W)\}.$$

To any automaton W there corresponds, up to labelling of states, a unique automaton $\min(W)$, called the *minimisation* of W , with $L(\min(W)) = L(W)$ and $\min(W)$ having the smallest possible number of states. We shall order the states s by $\lambda(s)$, giving us a unique minimisation. An automaton is *minimal* if it equals its own minimisation. The minimisation of W has one state for each distinct tail language of W , and is obtained by grouping together all states with the same tail language. See [AHU] Chapter 4 for details.

All automata considered will be assumed to be minimal as well as deterministic.

If $S \subseteq A^*$ and $k \in \mathbb{N}$ then

$$S_{\leq k} = \{w \in S : |w| \leq k\}$$

and is called a *truncation* or the *k-truncation* of S . Similarly, if W is an automaton then $W_{\leq k}$ is the minimal automaton accepting $L(W)_{\leq k}$.

Limits

Let $i \in \mathbb{N}$ and let W be an automaton. Recall that $W_{\leq k}$ is the minimal automaton accepting $L(W)_{\leq k}$. We shall call $(W_{\leq i})_{i=0}^{\infty}$ a sequence of automata tending to W and say W is the *limit* or the ∞ -*limit* of $(W_{\leq i})_{i=0}^{\infty}$. W is clearly uniquely specified by the $W_{\leq i}$ since they determine the whole of its language.

By extension, if $(W_{\leq i})_{i=0}^k$ is a finite subsequence at the start of this sequence, and M is an automaton with $M_{\leq i} = W_{\leq i}$ for $i = 0$ up to k then we shall say M is a *limit* or a *k-limit* of $(W_{\leq i})_{i=0}^k$.

Since knowledge of $W_{\leq k}$ implies knowledge of $W_{\leq i}$ for all i less than k , we may also equivalently talk of a *k-limit* of $W_{\leq k}$, which is any automaton M with $L(M)_{\leq k} = L(W_{\leq k})$.

Clearly for any finite sequence there are infinitely many possible continuations tending to different limits, as well as infinitely many continuations with no limit. If k is large enough, however, it may be that we know so much about W that any other k -limit would have to be much more complicated.

As an example, let W be the automaton whose language is $(a)^*$, i.e. the automaton with one state, an acceptor, and a single looping arrow labelled a . Then $L(W_{\leq k}) = \{\epsilon, a, aa, \dots, a^k\}$.

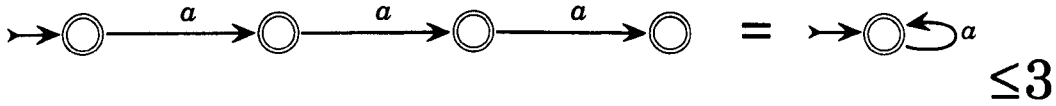


Diagram 7.2: $(a)^*$ truncated to depth 3

Any automaton that accepts all the powers of a up to a^n but not a^{n+1} must have at least $n + 1$ states. So any other k -limit of $W_{\leq k}$ has at least $k + 1$ states.

We shall say that an automaton W is the *natural k -limit* of an automaton $W_{\leq k}$ if $L(W)_{\leq k} = L(W_{\leq k})$ and W is the unique smallest automaton satisfying this condition.

We wish to determine when natural limits exist and how to find them. This falls into two parts: an algorithm that will determine the natural k -limit of $W_{\leq k}$ if such a thing exists, and a bound on k for a given W that will ensure that the natural k -limit of $W_{\leq k}$ is W .

Truncated Tails

Let $W_{\leq k}$ be the k -truncation of an automaton W . The *known tail* of a state s of $W_{\leq k}$ is the portion of the tail language of which we are certain, and is defined by

$$K(W_{\leq k}; s) = L(W_{\leq k}; s)_{\leq k - \delta(s)}.$$

Alternatively, it is those words w with $\lambda(s) \cdot w \in L(W_{\leq k})$.

If M is a limit of $W_{\leq k}$ then $K(W_{\leq k}; s)$ must be a truncation of the ‘true’ tail language $L(M; M[\lambda(s)])$ by virtue of the alternative definition above, and we say s is a *truncation* of $M[\lambda(s)]$.

Now since the natural limit (assuming it exists) will only have finitely many states, it can only have finitely many tail languages. If we take k sufficiently

large, we shall have lots of states of $W_{\leq k}$ each of which is a truncation of the same state in the natural limit.

There are, however, in general many more states in $W_{\leq k}$ than in its natural limit, so we will get a lot of duplication of these tail languages.

We shall therefore say that if $K(W_{\leq k}; s_1)$ is a truncation of $K(W_{\leq k}; s_2)$, then s_1 is a *known truncation* of s_2 . This will correspond to both s_1 and s_2 being truncations of the same state in the natural limit. We shall write this as $s_1 \prec s_2$. This gives us a partial ordering on the states of $W_{\leq k}$.

An example: Let W be the automaton accepting $(a)^*$ as before, and consider $W_{\leq 3}$, its 3-truncation. For $W_{\leq 3}$ all four states are known truncations of the start state and truncations of the single state of W .

We shall attempt to recover W by finding a representative of each of its states and connecting them together. The first part of this algorithm is finding the representatives – we shall use the maximal states under the ordering. It may be helpful to read the example in Chapter 9 in parallel with this section.

Algorithm

We shall obtain a set D of maximal states of $W_{\leq k}$, from which we shall construct W , in the following manner:

Start with $D = \emptyset$ and with two sets of states $todo = \{s_0\}$ and $done = \emptyset$.

Do the following for as long as $todo \neq \emptyset$:

Let s be the first state in $todo$.

If $\delta(s) > k$ then abandon the algorithm completely.

If for each $d \in D$, $s \not\prec d$ then:

Add s to D , and add all of its children that are not in $done$ to $todo$.

Remove s from $todo$ and add it to $done$.

If at any point in the process we find a state that is a known truncation of two different states in D (i.e. $s \prec d_1$ and $s \prec d_2$ for $d_1 \neq d_2 \in D$), then we shall abandon the algorithm for this k – we need more information to determine W uniquely.

There is a least depth δ such that no states of depth δ were added to D . The set of states of this depth is called the *terminal ring*. We shall assume that this layer is nonempty (equivalently, that $k > \delta(W)$), and abandon the algorithm if this is not so. Note that no states outside the terminal ring can be

added to D because we only ever add children of states in D to *todo*.

Arrows

Our guess at W , called X , will have D as its state set. It will have the same initial state as $W_{\leq k}$, and the same states will be accept states.

Consider an arrow of $W_{\leq k}$ labelled a leaving a state $s_1 \in D$ for a destination s_2 .

If $s_2 \in D$ then we add an arrow labelled a from s_1 to s_2 in X .

If $s_2 \notin D$ then s_2 is at most one deeper than s_1 , so is either in the terminal ring or inside it. s_2 has thus been examined by the first part of the algorithm and so $s_2 \prec d$ for some unique $d \in D$. We then add an arrow labelled a from s_1 to d in X .

The above algorithm gives us all the arrows for X , and leaves us with a fully specified automaton. We now show that if it runs to completion, it will give us the natural limit of $W_{\leq k}$; if it does not, then no natural limit exists. Moreover, if k is sufficiently large, it will give W as its answer.

Correctness

We want to show that X is still a k -truncation of W , i.e. that

$$L(X)_{\leq k} = L(W)_{\leq k} = L(W_{\leq k}).$$

Now for any state s of X , we can describe its tail language in terms of the destination states of its arrows:

$$L(X; s)_{\leq i} = \cup_{a \in A} a \cdot L(X; s[a])_{\leq i-1} \cup L(X; s)_{\leq 0}$$

By keeping the same states as accept states, we know that, for all states $s \in D$, we have $L(X; s)_{\leq 0} = L(W_{\leq k}; s)_{\leq 0}$.

We shall use induction. Assume that for some $r \in \mathbb{N}$ and for all $s \in D$,

$$L(X; s)_{\leq \min(r, k-\delta(s))} = L(W_{\leq k}; s)_{\leq \min(r, k-\delta(s))}$$

and pick $s \in D$ and try to prove the same equality for $r + 1$.

If $r + 1 > k - \delta(s)$ then the equality is trivially still true as the value of the minimum is unchanged. So we may assume $r + 1 \leq k - \delta(s)$, i.e. that $\delta(s) + 1 \leq k - r$.

For each letter $a \in A$, set $s_2 = W_{\leq k}[s][a]$.

If $s_2 \in D$ then $X[s][a] = s_2$ and since $r \leq k - (\delta(s) + 1)$, by hypothesis we have $L(X; s_2)_{\leq r} = L(W_{\leq k}; s_2)_{\leq r}$ and all is fine for the induction.

If $s_2 \notin D$, then set $d = X[s][a]$. Now, since $s \in D$, the arrow from s to s_2 has been processed by the second part of the algorithm, so d is the unique state in D with $s_2 \prec d$. Since $s \in D$, $\delta(s_2) \leq \delta(s) + 1$, so the known tail of s_2 has length at least $k - \delta(s) - 1$, which is at least r .

So $L(W_{\leq k}; s_2)_{\leq r} = L(W_{\leq k}; d)_{\leq r} = L(X; d)_{\leq r}$.

So, in either case, $L(W_{\leq k}; W_{\leq k}[s][a])_{\leq r} = L(X; X[s][a])_{\leq r}$.

So, since this is true for all $a \in A$,

$$L(W_{\leq k}; s)_{\leq r+1} = L(X; s)_{\leq r+1}$$

and the induction proof is complete.

Applying this to s_0 , which of course has depth 0, we get that

$$L(X; s_0)_{\leq k} = L(W_{\leq k}; s_0)_{\leq k}$$

and hence that

$$L(X)_{\leq k} = L(W_{\leq k})_{\leq k} = L(W)_{\leq k}.$$

Thus the X we have constructed is indeed a limit of $W_{\leq k}$.

Uniqueness and Minimality

We must now show that X is the unique smallest automaton that is a k -limit of $W_{\leq k}$, assuming that k is sufficiently large.

The number of states of a minimal finite state automaton is equal to the number of its tail languages. Since W has at least $|D|$ distinct known tails, W has at least $|D|$ tail languages, so has at least $|D|$ states.

Assuming we have the right number of states, there is necessarily only one way of wiring up the arrows (subject to ordering) – once we have decided which tail language a state represents, we know what languages are given by the states at the other ends of each of its arrows.

The automaton X , then, is the only automaton with $|D|$ states or fewer that is the k -limit of $W_{\leq k}$, so is the natural k -limit of $W_{\leq k}$.

Existence

In order to show that X exists and is the right answer, we must first show that the algorithm will run to completion for a sufficiently large k .

We define the *distinguishability* of two distinct states s_1 and s_2 of an automaton W to be

$$\rho(W; s_1, s_2) = \min\{|w| : w \in L(W; s_1) \triangle L(W; s_2)\}$$

i.e. the length of the shortest word that is acceptable from one of the states but not from the other.

We then define the distinguishability of an automaton as

$$\rho(W) = \max_{i \neq j} \rho(W; s_i, s_j)$$

so that any two states of W have tail languages that differ in their $\rho(W)$ -truncations.

Define the *reconstructibility* of an automaton to be $B(W) = \delta(W) + \rho(W) + 1$, and assume $k \geq B(W)$. Consider what happens when we run the algorithm on $W_{\leq k}$.

First we shall necessarily find a terminal ring. Consider the ring of states of depth $\delta(W) + 1$. All of them have known tail languages of length $\rho(W)$, which identify them with a unique state of W , and, via the shortest word leading to that state, with a state of D .

Thus, for all states s of $W_{\leq k}$ with $\delta(s) = \delta(W) + 1$, there is a unique state s' of W with $L(W; s')_{\leq \rho(W)} = L(W_{\leq k}; s)_{\leq \rho(W)}$. But we also know that $L(W; s')_{\leq \rho(W)} = L(W_{\leq k}; W_{\leq k}[\lambda(s')])_{\leq \rho(W)}$ and thus we get that s is a truncation of $W_{\leq k}[\lambda(s')]$.

Second, we shall never be in the situation of a state s inside or on the terminal ring being a truncation of two states s_1 and s_2 in D . For if this happens, $K(W_{\leq k}; s)$ is at least a $\rho(W)$ -truncation of a tail language from W , and as such specifies a unique state of W . But so are $K(W_{\leq k}; s_1)$ and $K(W_{\leq k}; s_2)$, and so they must all specify the same state.

So the algorithm will run to completion for $k \geq B(W)$, and X will be the natural k -limit of $W_{\leq k}$. The only remaining question is: does $X = W$?

By earlier remarks, this is true if and only if X and W have the same number of states. To each state s' of W there corresponds a state s of X given by the least word reaching s' in W . This state s will have depth at most that of s' , so has known tail of length at least $\rho(W) + 1$. But then it cannot be identified with any other state of W , so the correspondence is an injection, and hence X has at least as many states as W , so they are equal.

Thus, for $k \geq B(W)$, W is the natural k -limit of $W_{\leq k}$, and the algorithm will run to completion on $W_{\leq k}$, giving W .

Bounds

Trakhtenbrot and Barzdin in [TB] show that, for an automaton with n states over an alphabet A with $|A| \geq 2$,

$$\begin{aligned}\log_{|A|} n &< \delta + 1 \leq n \\ \log_{|A|} \log_2 n &< \rho + 1 \leq n\end{aligned}$$

[TB] also shows that for almost all automata W , $B(W) \leq C \log_{|A|} n$ where C is a constant. Almost all means here that there is an algorithm P that generates all automata with n states with equal probability and that

$$Prob(P \text{ makes } W \text{ with } B(W) \leq C \log_{|A|} n) \rightarrow 1 \text{ as } n \rightarrow \infty.$$

Checking correctness

We cannot, as yet, check that our automaton is correct once we have made it, though we know that at some point it must be. There are, however, some useful checks we can do that will throw out most wrong guesses.

First, we can calculate the maximum depth and distinguishability of the automaton we have made, and if the sum of these is larger than $k - 1$, we should be doubtful about our answer – it need not be incorrect, as some of the examples in Appendix 1 demonstrate, but it is worth trying with a larger k to be sure.

When using the algorithm to make a word acceptor, we do not need to start from $W_{\leq k}$, because this grows very large. Instead we can construct a partial word rejector as in Chapter 3, which is often much smaller, and on this we run the algorithm to get a limit. When the algorithm has produced an answer, we can then check to see if it accepts a sublanguage of the partial word rejector – it clearly must reject at least the words we know are unacceptable. We then take the complement of the answer to give us a word acceptor.

Further, when we make our partial word rejector, we ignore the rules we have read that have left-hand sides longer than k . They are nevertheless true, and we can check to see if the answer we have reached agrees with them. For global rules, the left-hand side must be rejected from each state; for local rules it need only be rejected from the start.

No examples of word acceptors have been found that pass all of these checks and yet are still wrong.

Chapter 8: Multipliers

Aim

This chapter will describe the construction of the multipliers from the Knuth–Bendix rules using the guessing algorithm. This is more complex than constructing the word acceptors, because both sides of the rules have to be used.

Two-tape Automata

As was described in Chapter 1, we use two-tape automata for the multipliers: the alphabet of M_a is $A' \times A'$ where $A' = A \cup \{\$ \}$. Here, $\$$ is the end-of string character, after which no other character can be read.

Although M_a actually accepts strings of the form $(a_1, b_1) \cdots (a_m, b_m)$, we shall abuse notation and write this as $(a_1 \cdot a_2 \cdots a_m, b_1 \cdot b_2 \cdots b_m)$. So M_a will accept strings (u, v) such that $u, v \in L(W)$ and $H\bar{u}a = H\bar{v}$, where $\bar{\$} = 1_G$.

To make things easier, we shall allow strings to end in more than one $(\$, \$)$ and still be acceptable – this does not change the essence of the language, but makes it easier to cope with left and right projections, as we shall see later.

Termination and Projections

The first thing to do is to make a terminated version of the word acceptor, that is, one which insists that the words it accepts end in $\$$.

We do this by adding two new states: one, an acceptor, for $\$$ to lead to from an old acceptor; a loop of $\* at that state, with everything else leading straight to the second new state, a fail state, which has a complete set of loops.

We then turn all the old accept states into fail states and add $\$$ arrows from them to the new acceptor.

This will give us the terminated word acceptor $W_\$$.

Because regular languages are closed under the operations of predicate calculus, if M is a two-tape automaton,

$$\text{Left}(M) = \{x : (\exists y \in A'^*)((x, y) \in L(M))\}$$

is also a regular language, as is $\text{Right}(M)$, and these are called the left and right *projections*.

Starting Concatenation Multiplier

It is certainly always true that if $v = u \cdot a$ is an acceptable word, then $(u, v) \in L(M_a)$, i.e. if it is acceptable to follow a word by a letter a , then it is acceptable to multiply that word by a just using concatenation.

We can use this to obtain a starting automaton. We consider the language

$$L_{\cdot a} = \{u \in L(W) : u \cdot a \in L(W)\}$$

This is regular because we are applying predicate calculus operations; alternatively, we are simply intersecting $L(M_a)$ with $\{(u, u \cdot a) : u \in A^*\}$.

We then construct the automaton that accepts the words of the form $(u, u \cdot a)$ for $u \in L_{\cdot a}$ – this will be our starting automaton for the next section.

Building Truncated Multipliers

In order to apply the guessing algorithm, we need $M_{a \leq n}$. We shall build this up in the following manner:

Find the first string $w\r with $w \in A^*$ that is acceptable to $W_\$,$ but does not yet appear as a left-hand side for our multiplier. Because we start with the concatenation multiplier, $w \cdot a$ cannot be acceptable, so must be reducible using some finite chain of Knuth–Bendix rules to some string u that is irreducible.

Any pair $(w\$^r, u\$^s)$ with $|w| + r = |u| + s$ and $r, s \geq 1$ must then be in the language of M_a . Moreover, if we have used no initial rules in reducing $w \cdot a$ to u , then we can make a broader statement: if $x \cdot w$ is acceptable to W and $x \cdot u$ is also acceptable, then $(x \cdot w\$^r, x \cdot u\$^s)$ must also be in $L(M_a)$.

We can thus make $w\r , and possibly many of the words ending in $w\r , be acceptable as a left-hand side of our multiplier, and then start again.

Whenever the length of $w\r increases, to n , say, then we know we have the correct multiplier truncated to $n - 1$, i.e. we have $M_{a \leq n-1}$. We can then use this to start the guessing algorithm.

Further, when we have produced a guess, we can apply a more stringent test to see if it is right: a correct multiplier must describe a bijection from $L(W)$ to itself. We can thus check that $\text{Left}(M_a) = \text{Right}(M_a) = L(W)$, and, further, that each word appears precisely once on both sides.

Chapter 9: Example

Aim

This section is a large worked example of the construction of a coset automatic system. The group used is the trefoil knot group $G = \langle a, b \mid aba = bab \rangle$ and the subgroup is $H = \langle a \rangle$.

Knuth–Bendix Rules

The following rules are generated to depth 8:

Ha	\Rightarrow	H	HA	\Rightarrow	H
aA	\Rightarrow	ϵ	Aa	\Rightarrow	ϵ
bB	\Rightarrow	ϵ	Bb	\Rightarrow	ϵ
bab	\Rightarrow	aba	baB	\Rightarrow	Aba
bAB	\Rightarrow	ABa	Bab	\Rightarrow	abA
BAb	\Rightarrow	aBA	BAB	\Rightarrow	ABA
$baaB$	\Rightarrow	$Abba$	$bAAB$	\Rightarrow	$ABBa$
$Baab$	\Rightarrow	$abbA$	$BAAb$	\Rightarrow	$aBBA$
$baaaB$	\Rightarrow	$Abbba$	$baaba$	\Rightarrow	$abaab$
$baabA$	\Rightarrow	$Abaab$	$bAAAB$	\Rightarrow	$ABBBa$
$Baaab$	\Rightarrow	$abbbA$	$BAAAb$	\Rightarrow	$aBBBA$
$BAABa$	\Rightarrow	$aBAAB$	$BAABA$	\Rightarrow	$ABAAB$
$baaaaB$	\Rightarrow	$Abbbba$	$baaaba$	\Rightarrow	$abaabb$
$baaabA$	\Rightarrow	$Abbaab$	$baabbA$	\Rightarrow	$Abaaab$
$bAAAAAB$	\Rightarrow	$ABBBBa$	$Baaaaab$	\Rightarrow	$abbbba$
$BAAAAAb$	\Rightarrow	$aBBBBa$	$BAAAABa$	\Rightarrow	$aBBBAAB$
$BAAABA$	\Rightarrow	$ABAABB$	$BAABBa$	\Rightarrow	$aBAAAB$
$baaaaaB$	\Rightarrow	$Abbbbbb$	$baaaaaba$	\Rightarrow	$abaabbb$
$baaaabA$	\Rightarrow	$Abbbbaab$	$baaabbA$	\Rightarrow	$Abbaaab$
$baabbbA$	\Rightarrow	$Abaaab$	$bAAAAAB$	\Rightarrow	$ABBBBba$
$Baaaaab$	\Rightarrow	$abbbbaA$	$BAAAAAb$	\Rightarrow	$aBBBBBA$
$BAAAABa$	\Rightarrow	$aBBBAAB$	$BAAABA$	\Rightarrow	$ABAABBB$
$BAAABBa$	\Rightarrow	$aBBAAAB$	$BAABBBa$	\Rightarrow	$aBAAAAAB$
$baaaaaaB$	\Rightarrow	$Abbbbbba$	$baaaaaba$	\Rightarrow	$abaabbbb$
$baaaaabA$	\Rightarrow	$Abbbbbaab$	$baaaabbA$	\Rightarrow	$Abbbbaab$
$baaabbbA$	\Rightarrow	$Abbaaab$	$baabbbbA$	\Rightarrow	$Abaaaaaab$
$bAAAAAAB$	\Rightarrow	$ABBBBBBa$	$Baaaaaab$	\Rightarrow	$abbbbbba$
$BAAAAAAb$	\Rightarrow	$aBBBBBBA$	$BAAAAABa$	\Rightarrow	$aBBBBBAAB$
$BAAAAABA$	\Rightarrow	$ABAABBBB$	$BAAABBBa$	\Rightarrow	$aBBBBAAAB$
$BAAABBBa$	\Rightarrow	$aBBAAAAAB$	$BAABBBBa$	\Rightarrow	$aBAAAAAAB$

Starting Automaton

The above rules give us the starting automaton on which to run the guessing algorithm. It is:

	a	A	b	B		a	A	b	B		a	A	b	B			
1	A	2	2	3	4;	2	N	2	2	2	2;	3	A	5	6	3	2;
4	A	7	8	2	4;	5	A	9	2	2	2;	6	A	2	10	3	2;
7	A	11	2	2	4;	8	A	2	12	2	2;	9	A	13	2	14	2;
10	A	2	15	3	2;	11	A	16	2	2	4;	12	A	2	17	2	18;
13	A	19	2	20	2;	14	A	2	2	21	2;	15	A	2	22	3	2;
16	A	23	2	2	4;	17	A	2	24	2	25;	18	A	2	2	2	26;
19	A	27	2	28	2;	20	A	2	2	29	2;	21	A	5	2	29	2;
22	A	2	30	3	2;	23	A	31	2	2	4;	24	A	2	32	2	33;
25	A	2	2	2	34;	26	A	2	8	2	34;	27	A	35	2	36	2;
28	A	2	2	37	2;	29	A	5	2	37	2;	30	A	2	38	3	2;
31	A	39	2	2	4;	32	A	2	40	2	41;	33	A	2	2	2	42;
34	A	2	8	2	42;	35	A	43	2	3	2;	36	A	2	2	3	2;
37	A	5	2	3	2;	38	A	2	44	3	2;	39	A	43	2	2	4;
40	A	2	44	2	4;	41	A	2	2	2	4;	42	A	2	8	2	4;
43	A	43	2	3	4;	44	A	2	44	3	4;						

Guessing

We can now begin to guess the word acceptor based on the above data.

Here is a list of the tail languages of some of the states of the above automaton, truncated at a length of 2:

State	Depth	0	1	2	Children
1	0	ϵ	b, B	ba, bA, bb, Ba, BA, BB	2, 3, 4
2	1				2
3	1	ϵ	a, A, b	aa, AA, Ab, ba, bA, bb	2, 3, 5, 6
4	1	ϵ	a, A, B	aa, aB, AA, Ba, BA, BB	2, 4, 7, 8
5	2	ϵ	a	aa, ab	2, 9
6	2	ϵ	A, b	AA, Ab, ba, bA, bb	2, 3, 10
7	2	ϵ	a, B	aa, aB, Ba, BA, BB	2, 4, 11
8	2	ϵ	A	AA, AB	2, 12
9	3	ϵ	a, b	aa, ab, bb	2, 13, 14
10	3	ϵ	A, b	AA, Ab, ba, bA, bb	=6
11	3	ϵ	a, B	aa, aB, Ba, BA, BB	=7
12	3	ϵ	A, B	AA, AB, BB	2, 17, 18
13	4	ϵ	a, b	aa, ab, bb	=9
14	4	ϵ	b	ba, bb	2, 21
17	4	ϵ	A, B	AA, AB, BB	=12
18	4	ϵ	B	BA, BB	2, 26
21	5	ϵ	a, b	aa, ba, bb	2, 5, 29
26	5	ϵ	A, B	AA, BA, BB	2, 8, 34
29	6	ϵ	a, b	aa, ba, bb	=21
34	6	ϵ	A, B	AA, BA, BB	=26

As the algorithm runs through this automaton, it will find states 1 to 9 to be distinguishable from each other: they all have differing tail languages by length 2, and since they are all of depth at most 3, this is well inside the known depth of 8.

State 10 is different, however. From the above list it can be seen that it is indistinguishable from state 6 to length 2; in fact it is indistinguishable all the way to length 8 – $\delta(s_6) = 5$. We thus identify state 10 with state 6 and ignore any of its children.

The same is true of state 11, which identifies with state 7; state 12 is distinguishable from all previous states, however.

Continuing with the children of states 9 and 12, we find that 13 is indistinguishable from 9 to depth 8 – $\delta(s_{13}) = 4$, and similarly for states 17 and 12; states 14 and 18 survive.

We do not consider states 15 and 16 since they are children of states that we have already decided to identify with earlier states, and so must themselves be identified – in fact they will identify with their parent states 10 and 11 and hence states 6 and 7.

We get two new states to look at at depth 5: 21 and 26.

State 29, which is state 21's child of depth 6, is identical to state 21 to length $8 - 6 = 2$, so gets identified with it; similarly with states 34 and 26.

We now have a terminal ring: the only states of interest of depth 6 all identify with earlier states. We have finished.

The resulting automaton is the following:

	a	A	b	B		a	A	b	B		a	A	b	B			
1	A	2	2	3	4;	2	N	2	2	2	2;	3	A	5	6	3	2;
4	A	7	8	2	4;	5	A	9	2	2	2;	6	A	2	6	3	2;
7	A	7	2	2	4;	8	A	2	10	2	2;	9	A	9	2	11	2;
10	A	2	10	2	12;	11	A	2	2	13	2;	12	A	2	2	2	14;
13	A	5	2	13	2;	14	A	2	8	2	14;						

It has maximum depth 5 and distinguishability 2, so has reconstructibility 8 – the length to which the rule set is complete. It also accepts a sublanguage of the first automaton, which is sensible, because it will reject more strings.

This automaton is the correct one. This can either be checked by running the Knuth–Bendix for longer and confirming that it does indeed give the right answer for greater depths, or by constructing the multipliers and checking that the system works – only the latter will give us a conclusive proof.

Multipliers

We can also make the multipliers using the methods from the previous section.

They all require rules to depth 10 or 11, and are shown here. Note that, because multipliers for this system would in general have 25 arrows coming out of each state, most of which go to a terminal fail state, an abbreviated printing method is used. The most common destination state is called the default state, and arrows to it are omitted; this state is indeed the terminal fail state.

Notice that, unlike in automatic groups, the multiplier need not be fixed-point free – for example, M_a accepts the string $(\$, \$)$.

M_a :

- 1 N $(b,b) \rightarrow 3 (B,B) \rightarrow 4 (\$, \$) \rightarrow 5$;
- 2 N (default);
- 3 N $(a,a) \rightarrow 6 (A,A) \rightarrow 7 (A,b) \rightarrow 8 (A,\$) \rightarrow 9 (b,a) \rightarrow 10 (b,b) \rightarrow 11 (\$,a) \rightarrow 9$;
- 4 N $(a,a) \rightarrow 12 (A,A) \rightarrow 13 (A,B) \rightarrow 14 (A,\$) \rightarrow 9 (B,a) \rightarrow 15 (B,B) \rightarrow 16 (\$,a) \rightarrow 9$;
- 5 A $(\$, \$) \rightarrow 5$;
- 6 N $(a,a) \rightarrow 17 (\$,a) \rightarrow 9$;
- 7 N $(A,A) \rightarrow 7 (A,b) \rightarrow 8 (A,\$) \rightarrow 9 (b,b) \rightarrow 11$;
- 8 N $(b,a) \rightarrow 18$;
- 9 N $(\$, \$) \rightarrow 5$;
- 10 N $(a,a) \rightarrow 19 (b,a) \rightarrow 10$;
- 11 N $(a,a) \rightarrow 20 (A,A) \rightarrow 7 (A,b) \rightarrow 8 (A,\$) \rightarrow 9 (b,b) \rightarrow 11 (\$,a) \rightarrow 9$;
- 12 N $(a,a) \rightarrow 12 (B,a) \rightarrow 15 (B,B) \rightarrow 16 (\$,a) \rightarrow 9$;
- 13 N $(A,A) \rightarrow 21 (A,\$) \rightarrow 9$;
- 14 N $(A,A) \rightarrow 22 (A,B) \rightarrow 14$;
- 15 N $(A,B) \rightarrow 23$;
- 16 N $(a,a) \rightarrow 12 (A,A) \rightarrow 24 (A,\$) \rightarrow 9 (B,a) \rightarrow 15 (B,B) \rightarrow 16 (\$,a) \rightarrow 9$;
- 17 N $(a,a) \rightarrow 25 (a,b) \rightarrow 26 (b,b) \rightarrow 27 (\$,a) \rightarrow 9$;
- 18 N $(a,a) \rightarrow 28 (b,a) \rightarrow 18$;
- 19 N $(a,a) \rightarrow 26$;
- 20 N $(a,a) \rightarrow 25 (\$,a) \rightarrow 9$;
- 21 N $(A,A) \rightarrow 29 (A,\$) \rightarrow 9 (B,A) \rightarrow 30 (B,B) \rightarrow 31$;
- 22 N $(A,A) \rightarrow 30$;
- 23 N $(A,A) \rightarrow 32 (A,B) \rightarrow 23$;
- 24 N $(A,A) \rightarrow 29 (A,\$) \rightarrow 9$;
- 25 N $(a,a) \rightarrow 25 (b,b) \rightarrow 33 (\$,a) \rightarrow 9$;
- 26 N $(a,b) \rightarrow 26 (b,b) \rightarrow 34$;
- 27 N $(b,b) \rightarrow 35 (\$, \$) \rightarrow 5$;
- 28 N $(a,b) \rightarrow 36$;
- 29 N $(A,A) \rightarrow 29 (A,\$) \rightarrow 9 (B,B) \rightarrow 37$;
- 30 N $(B,A) \rightarrow 30 (B,B) \rightarrow 38$;
- 31 N $(B,B) \rightarrow 39 (\$, \$) \rightarrow 5$;
- 32 N $(B,A) \rightarrow 40$;
- 33 N $(b,b) \rightarrow 41$;
- 34 N $(b,b) \rightarrow 10 (\$, \$) \rightarrow 5$;
- 35 N $(a,a) \rightarrow 6 (b,a) \rightarrow 10 (b,b) \rightarrow 41 (\$,a) \rightarrow 9$;
- 36 N $(a,b) \rightarrow 36 (b,b) \rightarrow 8 (b,\$) \rightarrow 9$;
- 37 N $(B,B) \rightarrow 42$;
- 38 N $(B,B) \rightarrow 14 (\$, \$) \rightarrow 5$;
- 39 N $(A,A) \rightarrow 13 (A,B) \rightarrow 14 (A,\$) \rightarrow 9 (B,B) \rightarrow 42$;
- 40 N $(B,A) \rightarrow 40 (B,B) \rightarrow 15 (\$,B) \rightarrow 9$;
- 41 N $(a,a) \rightarrow 20 (b,b) \rightarrow 41 (\$,a) \rightarrow 9$;
- 42 N $(A,A) \rightarrow 24 (A,\$) \rightarrow 9 (B,B) \rightarrow 42$;

M_A :

- 1 N $(b,b) \rightarrow 3 (B,B) \rightarrow 4 (\$, \$) \rightarrow 5$;
- 2 N (default);
- 3 N $(a,a) \rightarrow 6 (a,b) \rightarrow 7 (a,\$) \rightarrow 8 (A,A) \rightarrow 9 (b,A) \rightarrow 10 (b,b) \rightarrow 11 (\$,A) \rightarrow 8$;
- 4 N $(a,a) \rightarrow 12 (a,B) \rightarrow 13 (a,\$) \rightarrow 8 (A,A) \rightarrow 14 (B,A) \rightarrow 15 (B,B) \rightarrow 16 (\$,A) \rightarrow 8$;
- 5 A $(\$, \$) \rightarrow 5$;
- 6 N $(a,a) \rightarrow 17 (a,\$) \rightarrow 8$;
- 7 N $(a,a) \rightarrow 18 (a,b) \rightarrow 7$;
- 8 N $(\$, \$) \rightarrow 5$;
- 9 N $(A,A) \rightarrow 9 (b,A) \rightarrow 10 (b,b) \rightarrow 11 (\$,A) \rightarrow 8$;
- 10 N $(a,b) \rightarrow 19$;
- 11 N $(a,a) \rightarrow 20 (a,\$) \rightarrow 8 (A,A) \rightarrow 9 (b,A) \rightarrow 10 (b,b) \rightarrow 11 (\$,A) \rightarrow 8$;
- 12 N $(a,a) \rightarrow 12 (a,B) \rightarrow 13 (a,\$) \rightarrow 8 (B,B) \rightarrow 16$;
- 13 N $(B,A) \rightarrow 21$;
- 14 N $(A,A) \rightarrow 22 (\$,A) \rightarrow 8$;
- 15 N $(A,A) \rightarrow 23 (B,A) \rightarrow 15$;
- 16 N $(a,a) \rightarrow 12 (a,B) \rightarrow 13 (a,\$) \rightarrow 8 (A,A) \rightarrow 24 (B,B) \rightarrow 16 (\$,A) \rightarrow 8$;
- 17 N $(a,a) \rightarrow 25 (a,\$) \rightarrow 8 (b,a) \rightarrow 26 (b,b) \rightarrow 27$;
- 18 N $(a,a) \rightarrow 26$;
- 19 N $(a,a) \rightarrow 28 (a,b) \rightarrow 19$;
- 20 N $(a,a) \rightarrow 25 (a,\$) \rightarrow 8$;
- 21 N $(A,A) \rightarrow 29 (B,A) \rightarrow 21$;
- 22 N $(A,A) \rightarrow 30 (A,B) \rightarrow 31 (B,B) \rightarrow 32 (\$,A) \rightarrow 8$;
- 23 N $(A,A) \rightarrow 31$;
- 24 N $(A,A) \rightarrow 30 (\$,A) \rightarrow 8$;
- 25 N $(a,a) \rightarrow 25 (a,\$) \rightarrow 8 (b,b) \rightarrow 33$;
- 26 N $(b,a) \rightarrow 26 (b,b) \rightarrow 34$;
- 27 N $(b,b) \rightarrow 35 (\$, \$) \rightarrow 5$;
- 28 N $(b,a) \rightarrow 36$;
- 29 N $(A,B) \rightarrow 37$;
- 30 N $(A,A) \rightarrow 30 (B,B) \rightarrow 38 (\$,A) \rightarrow 8$;
- 31 N $(A,B) \rightarrow 31 (B,B) \rightarrow 39$;
- 32 N $(B,B) \rightarrow 40 (\$, \$) \rightarrow 5$;
- 33 N $(b,b) \rightarrow 41$;
- 34 N $(b,b) \rightarrow 7 (\$, \$) \rightarrow 5$;
- 35 N $(a,a) \rightarrow 6 (a,b) \rightarrow 7 (a,\$) \rightarrow 8 (b,b) \rightarrow 41$;
- 36 N $(b,a) \rightarrow 36 (b,b) \rightarrow 10 (\$,b) \rightarrow 8$;
- 37 N $(A,B) \rightarrow 37 (B,B) \rightarrow 13 (B,\$) \rightarrow 8$;
- 38 N $(B,B) \rightarrow 42$;
- 39 N $(B,B) \rightarrow 15 (\$, \$) \rightarrow 5$;
- 40 N $(A,A) \rightarrow 14 (B,A) \rightarrow 15 (B,B) \rightarrow 42 (\$,A) \rightarrow 8$;
- 41 N $(a,a) \rightarrow 20 (a,\$) \rightarrow 8 (b,b) \rightarrow 41$;
- 42 N $(A,A) \rightarrow 24 (B,B) \rightarrow 42 (\$,A) \rightarrow 8$;

M_b :

- 1 N $(b,b) \rightarrow 3 (B,b) \rightarrow 4 (B,B) \rightarrow 5 (B,\$) \rightarrow 6 (\$,b) \rightarrow 6$;
- 2 N (default);
- 3 N $(a,a) \rightarrow 7 (A,A) \rightarrow 8 (A,b) \rightarrow 9 (b,a) \rightarrow 10 (b,b) \rightarrow 11 (\$,b) \rightarrow 6$;
- 4 N $(a,A) \rightarrow 6 (a,b) \rightarrow 4 (B,A) \rightarrow 4$;
- 5 N $(a,a) \rightarrow 12 (A,A) \rightarrow 13 (A,B) \rightarrow 14 (B,a) \rightarrow 15 (B,B) \rightarrow 16 (B,\$) \rightarrow 6$;
- 6 N $(\$, \$) \rightarrow 17$;
- 7 N $(a,a) \rightarrow 18 (\$, \$) \rightarrow 17$;
- 8 N $(A,A) \rightarrow 8 (A,b) \rightarrow 9 (b,b) \rightarrow 11 (\$,b) \rightarrow 6$;
- 9 N $(b,a) \rightarrow 19$;
- 10 N $(a,a) \rightarrow 20 (b,a) \rightarrow 10$;
- 11 N $(a,a) \rightarrow 21 (A,A) \rightarrow 8 (A,b) \rightarrow 9 (b,b) \rightarrow 11 (\$,b) \rightarrow 6$;
- 12 N $(a,a) \rightarrow 12 (B,a) \rightarrow 15 (B,B) \rightarrow 16 (B,\$) \rightarrow 6$;
- 13 N $(A,A) \rightarrow 22 (\$, \$) \rightarrow 17$;
- 14 N $(A,A) \rightarrow 23 (A,B) \rightarrow 14$;
- 15 N $(A,B) \rightarrow 24$;
- 16 N $(a,a) \rightarrow 12 (A,A) \rightarrow 25 (B,a) \rightarrow 15 (B,B) \rightarrow 16 (B,\$) \rightarrow 6$;
- 17 A $(\$, \$) \rightarrow 17$;
- 18 N $(a,a) \rightarrow 26 (a,b) \rightarrow 27 (b,b) \rightarrow 28 (\$,b) \rightarrow 6$;
- 19 N $(a,a) \rightarrow 29 (a,\$) \rightarrow 6 (b,a) \rightarrow 19$;
- 20 N $(a,a) \rightarrow 27 (\$, \$) \rightarrow 17$;
- 21 N $(a,a) \rightarrow 26$;
- 22 N $(A,A) \rightarrow 30 (B,A) \rightarrow 31 (B,B) \rightarrow 32 (B,\$) \rightarrow 6$;
- 23 N $(A,A) \rightarrow 31 (\$, \$) \rightarrow 17$;
- 24 N $(A,A) \rightarrow 33 (A,B) \rightarrow 24 (\$,A) \rightarrow 6$;
- 25 N $(A,A) \rightarrow 30$;
- 26 N $(a,a) \rightarrow 26 (b,b) \rightarrow 34 (\$,b) \rightarrow 6$;
- 27 N $(a,b) \rightarrow 27 (b,b) \rightarrow 35$;
- 28 N $(b,b) \rightarrow 36 (\$,b) \rightarrow 6$;
- 29 N $(a,b) \rightarrow 37$;
- 30 N $(A,A) \rightarrow 30 (B,B) \rightarrow 38 (B,\$) \rightarrow 6$;
- 31 N $(B,A) \rightarrow 31 (B,B) \rightarrow 39$;
- 32 N $(B,B) \rightarrow 40 (B,\$) \rightarrow 6$;
- 33 N $(B,A) \rightarrow 41$;
- 34 N $(b,b) \rightarrow 42 (\$,b) \rightarrow 6$;
- 35 N $(b,b) \rightarrow 10$;
- 36 N $(a,a) \rightarrow 7 (b,a) \rightarrow 10 (b,b) \rightarrow 42 (\$,b) \rightarrow 6$;
- 37 N $(a,b) \rightarrow 37 (b,b) \rightarrow 9$;
- 38 N $(B,B) \rightarrow 43 (B,\$) \rightarrow 6$;
- 39 N $(B,B) \rightarrow 14$;
- 40 N $(A,A) \rightarrow 13 (A,B) \rightarrow 14 (B,B) \rightarrow 43 (B,\$) \rightarrow 6$;
- 41 N $(B,A) \rightarrow 41 (B,B) \rightarrow 15$;
- 42 N $(a,a) \rightarrow 21 (b,b) \rightarrow 42 (\$,b) \rightarrow 6$;
- 43 N $(A,A) \rightarrow 25 (B,B) \rightarrow 43 (B,\$) \rightarrow 6$;

M_B :

- 1 N $(b,b) \rightarrow 3 (b,B) \rightarrow 4 (b,\$) \rightarrow 5 (B,B) \rightarrow 6 (\$,B) \rightarrow 5$;
- 2 N (default);
- 3 N $(a,a) \rightarrow 7 (a,b) \rightarrow 8 (A,A) \rightarrow 9 (b,A) \rightarrow 10 (b,b) \rightarrow 11 (b,\$) \rightarrow 5$;
- 4 N $(A,a) \rightarrow 5 (A,B) \rightarrow 4 (b,a) \rightarrow 4$;
- 5 N $(\$, \$) \rightarrow 12$;
- 6 N $(a,a) \rightarrow 13 (a,B) \rightarrow 14 (A,A) \rightarrow 15 (B,A) \rightarrow 16 (B,B) \rightarrow 17 (\$,B) \rightarrow 5$;
- 7 N $(a,a) \rightarrow 18 (\$, \$) \rightarrow 12$;
- 8 N $(a,a) \rightarrow 19 (a,b) \rightarrow 8$;
- 9 N $(A,A) \rightarrow 9 (b,A) \rightarrow 10 (b,b) \rightarrow 11 (b,\$) \rightarrow 5$;
- 10 N $(a,b) \rightarrow 20$;
- 11 N $(a,a) \rightarrow 21 (A,A) \rightarrow 9 (b,A) \rightarrow 10 (b,b) \rightarrow 11 (b,\$) \rightarrow 5$;
- 12 A $(\$, \$) \rightarrow 12$;
- 13 N $(a,a) \rightarrow 13 (a,B) \rightarrow 14 (B,B) \rightarrow 17 (\$,B) \rightarrow 5$;
- 14 N $(B,A) \rightarrow 22$;
- 15 N $(A,A) \rightarrow 23 (\$, \$) \rightarrow 12$;
- 16 N $(A,A) \rightarrow 24 (B,A) \rightarrow 16$;
- 17 N $(a,a) \rightarrow 13 (a,B) \rightarrow 14 (A,A) \rightarrow 25 (B,B) \rightarrow 17 (\$,B) \rightarrow 5$;
- 18 N $(a,a) \rightarrow 26 (b,a) \rightarrow 27 (b,b) \rightarrow 28 (b,\$) \rightarrow 5$;
- 19 N $(a,a) \rightarrow 27 (\$, \$) \rightarrow 12$;
- 20 N $(a,a) \rightarrow 29 (a,b) \rightarrow 20 (\$,a) \rightarrow 5$;
- 21 N $(a,a) \rightarrow 26$;
- 22 N $(A,A) \rightarrow 30 (A,\$) \rightarrow 5 (B,A) \rightarrow 22$;
- 23 N $(A,A) \rightarrow 31 (A,B) \rightarrow 32 (B,B) \rightarrow 33 (\$,B) \rightarrow 5$;
- 24 N $(A,A) \rightarrow 32 (\$, \$) \rightarrow 12$;
- 25 N $(A,A) \rightarrow 31$;
- 26 N $(a,a) \rightarrow 26 (b,b) \rightarrow 34 (b,\$) \rightarrow 5$;
- 27 N $(b,a) \rightarrow 27 (b,b) \rightarrow 35$;
- 28 N $(b,b) \rightarrow 36 (b,\$) \rightarrow 5$;
- 29 N $(b,a) \rightarrow 37$;
- 30 N $(A,B) \rightarrow 38$;
- 31 N $(A,A) \rightarrow 31 (B,B) \rightarrow 39 (\$,B) \rightarrow 5$;
- 32 N $(A,B) \rightarrow 32 (B,B) \rightarrow 40$;
- 33 N $(B,B) \rightarrow 41 (\$,B) \rightarrow 5$;
- 34 N $(b,b) \rightarrow 42 (b,\$) \rightarrow 5$;
- 35 N $(b,b) \rightarrow 8$;
- 36 N $(a,a) \rightarrow 7 (a,b) \rightarrow 8 (b,b) \rightarrow 42 (b,\$) \rightarrow 5$;
- 37 N $(b,a) \rightarrow 37 (b,b) \rightarrow 10$;
- 38 N $(A,B) \rightarrow 38 (B,B) \rightarrow 14$;
- 39 N $(B,B) \rightarrow 43 (\$,B) \rightarrow 5$;
- 40 N $(B,B) \rightarrow 16$;
- 41 N $(A,A) \rightarrow 15 (B,A) \rightarrow 16 (B,B) \rightarrow 43 (\$,B) \rightarrow 5$;
- 42 N $(a,a) \rightarrow 21 (b,b) \rightarrow 42 (b,\$) \rightarrow 5$;
- 43 N $(A,A) \rightarrow 25 (B,B) \rightarrow 43 (\$,B) \rightarrow 5$;

Chapter 10: Existence

Aim

This chapter shows that quasiconvex subgroups of hyperbolic groups are coset automatic. For further information about hyperbolic groups and hyperbolic spaces, see [GdelaH].

Definitions

Let (X, d) be a metric space; we shall write $|x - y|$ for $d(x, y)$.

Let x, y be two points in X and let $a = |x - y|$. A *geodesic* in X from x to y is an isometry $g: [0, a] \rightarrow X$ with $g(0) = x$ and $g(a) = y$; its path, $[g]$, is its range as a function. X is a *geodesic space* if for all pairs of points $x, y \in X$ there is a geodesic $[0, |x - y|] \rightarrow X$ from x to y – this geodesic need not be unique.

A *geodesic triangle* with vertices x, y and z is the union of three geodesics joining the vertices in pairs. Degenerate cases in which, say, $y = z$ are allowed.

Although two points x and y in a geodesic space X do not in general determine a unique geodesic, it is nevertheless convenient to write $[x, y]$ to mean an arbitrary geodesic from x to y .

Given a number $\delta \geq 0$, a geodesic metric space satisfies the *Rips condition* with constant δ if for all geodesic triangles Δ in X the distance from a point on one side of Δ to the union of the other two sides is at most δ . In symbols, for all $\Delta = [x, y] \cup [y, z] \cup [z, x]$, and for all $u \in [y, z]$, $d(u, [x, y] \cup [z, x]) \leq \delta$.

A geodesic metric space X is δ -*hyperbolic* if it satisfies the Rips condition for some δ ; this is sometimes called *word hyperbolic* or just *hyperbolic*.

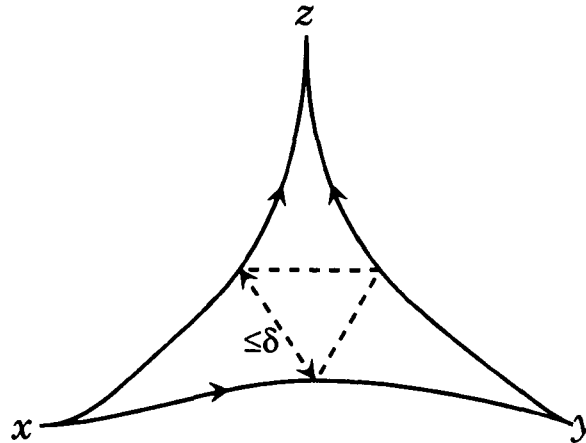


Diagram 10.1: Hyperbolic space

Examples and Useful Facts

These examples are shown to be hyperbolic in [GdelaH].

Any tree is 0-hyperbolic.

For any two quasi-isometric spaces, if one is hyperbolic, so is the other (though not necessarily with the same δ). Accordingly, a group is hyperbolic if its Cayley graph with respect to some set of generators is hyperbolic.

A finitely generated free group is 0-hyperbolic.

The fundamental group of a Riemannian manifold with negative curvature is hyperbolic.

A group satisfying the small cancellation hypothesis $C'(1/6)$ or one satisfying $C'(1/4)$ and $T(4)$ is hyperbolic.

Left-boundedness

Let H be a subgroup of G , a δ -hyperbolic group. An H -geodesic g is a geodesic in G such that for $0 \leq t \leq |g|$, $d(g(t), H) = t$. That is, $g(t)$ is a shortest word from H to $Hg(t)$. Note that g is necessarily also a geodesic, so that for all $h \in H$,

$$d(hg(t), H) = d(g(t), H) = d(g(t), 1) = |g(t)| = t$$

and I shall use these equalities without comment.

A *geodesic string* w is a string in A^* with \overline{w} a geodesic.

If w_1 and w_2 are H -geodesic strings and $\overline{w_1 a} = h \overline{w_2}$ for some $a \in A$, then we have the *extended Rips condition* that for all t , $d(\overline{w_1(t)}, [h] \cup h[\overline{w_2}]) \leq 2\delta$, $d(h \overline{w_2(t)}, [h] \cup [w_1 a]) \leq 2\delta$ and $d(h(t), [\overline{w_1}] \cup h[\overline{w_2}]) \leq 2\delta$ – see Diagram 10.2 on the next page.

To prove this, we draw in the two diagonal geodesics u and v across the quadrilateral. For $\overline{w_1(t)}$, for example, either $\overline{w_1(t)}$ is within δ of the a across the top, or it is within δ of the geodesic u from 1 to $\overline{w_1 a}$. Any point on u , using the triangle (u, h, w_2) , is within δ of either $[h]$ or $h[\overline{w_2}]$. So $\overline{w_1(t)}$ is within 2δ of the other two sides. Similarly for w_2 , with v replacing u .

For h , note that either $h(t)$ is within δ of $h[\overline{w_2}]$ or it is within δ of u . Any point on u is within δ of $[\overline{w_1}]$ or the a along the top. But being within δ of the a necessitates being within δ of one of the two ends, both of which are points on the sides we want to be within δ of. Thus any point on $[h]$ is within 2δ of one of the two sides.

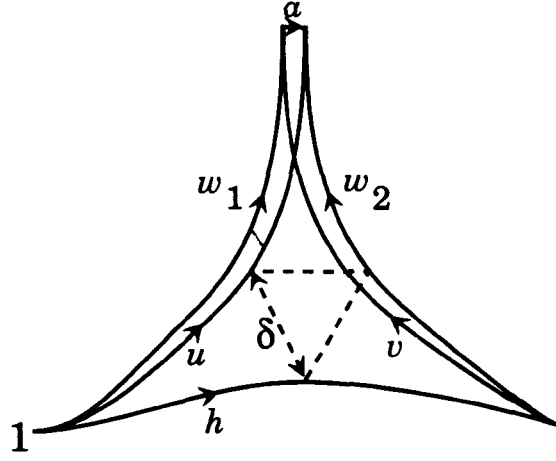


Diagram 10.2: Extended Rips condition

If $H\overline{w_1a} = H\overline{w_2}$ then there is some unique $h \in H$ with $\overline{w_1a} = h\overline{w_2}$; here $h = \overline{w_1a} \cdot \overline{w_2}^{-1}$. A subgroup $H \leq G$ is ε -left-bounded if for some $\varepsilon \geq 0$ and for all H -geodesic strings w_1, w_2 and all $a \in A \cup \{\epsilon\}$, $|\overline{w_1a} \cdot \overline{w_2}^{-1}| \leq \varepsilon$.

Early Take-off

Let G be a finitely generated δ -hyperbolic group and let H be an ε -left-bounded subgroup. G will have the word-metric d .

Let w_1 and w_2 be H -geodesic strings with $H\overline{w_1a} = H\overline{w_2}$; there is thus some $h \in H$ with $\overline{w_1a} = h\overline{w_2}$ and $|h| \leq \varepsilon$.

Since w_2 is an H -geodesic string, for all $0 \leq t \leq |w_2|$, $d(\overline{w_2(t)}, H) = t$.

Now,

$$d(h\overline{w_2(t)}, H) = d(\overline{w_2(t)}, H) = t$$

so

$$t = d(h\overline{w_2(t)}, H) \leq d(h\overline{w_2(t)}, [h]) + |h|.$$

So, as $|h| \leq \varepsilon$, if $t > 2\delta + \varepsilon$, $d(h\overline{w_2(t)}, [h]) > 2\delta$. Similarly $d(\overline{w_1(t)}, [h]) > 2\delta$.

Thus, using the extended Rips condition, we have both $d(h\overline{w_2(t)}, \overline{w_1a}) \leq 2\delta$ and $d(\overline{w_1(t)}, h\overline{w_2}) \leq 2\delta$.

So after length $2\delta + \varepsilon$ both w_1 and w_2 have left h behind and are close to each other. We thus have the following picture:

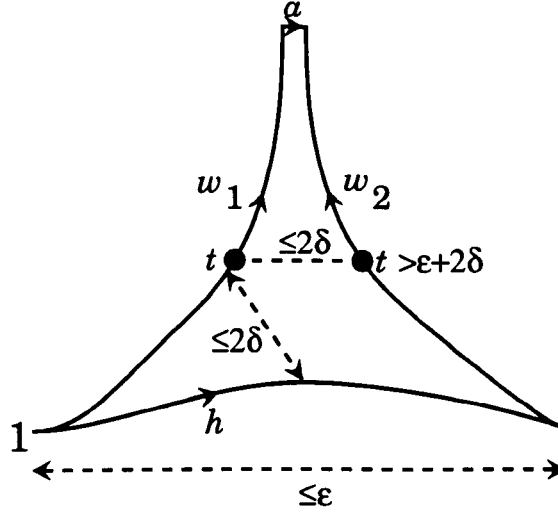


Diagram 10.3: Early take-off

The subgroup therefore affects only the first $2\delta + \varepsilon$ letters of the acceptable words after which they obey the ordinary closeness property of words in an automatic group.

Parallelism

We want to show that w_1 and w_2 travel approximately side-by-side as they head away from H towards their meeting point.

Suppose for some t , $d(\overline{w_1(t)}, h[\overline{w_2}]) \leq 2\delta$, i.e. the two paths come within 2δ – see Diagram 10.4. So there must be some s with $d(\overline{w_1(t)}, h\overline{w_2(s)}) \leq 2\delta$.

Now, $d(\overline{w_1(t)}, h\overline{w_2(s)}) = d(h^{-1}\overline{w_1(t)}, \overline{w_2(s)})$.

Let $r = |h^{-1}\overline{w_1(t)}|$. We have the following picture:

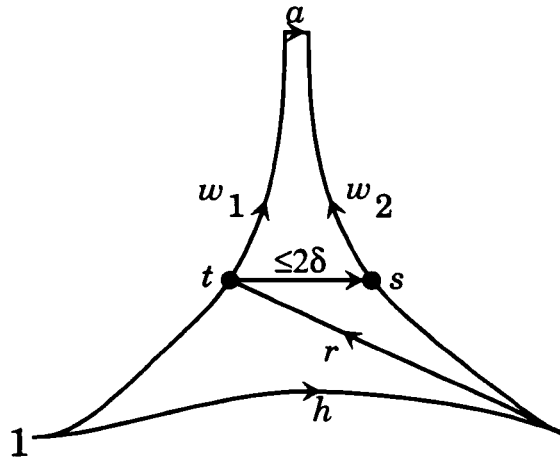


Diagram 10.4: Parallelism

Since w_1 and w_2 are geodesics, from the triangle $(1, h, \overline{w_1(t)})$, $|r - t| \leq \varepsilon$

and from the triangle $(h, \overline{hw_2(s)}, \overline{w_1(t)})$, $|r - s| \leq 2\delta$. So $|t - s| \leq \varepsilon + 2\delta$.

So

$$d(\overline{w_1(t)}, \overline{hw_2(t)}) \leq 2\delta + |t - s| \leq 4\delta + \varepsilon.$$

Similarly, if for some t , $d(\overline{hw_2(t)}, [w_1]) \leq 2\delta$ then again

$$d(\overline{w_1(t)}, \overline{hw_2(t)}) \leq 2\delta + |t - s| \leq 4\delta + \varepsilon.$$

Alternatively, if, for some t , we have are in a situation where neither of $d(\overline{w_1(t)}, h\overline{[w_2]}) \leq 2\delta$ or $d(\overline{hw_2(t)}, \overline{[w_1a]}) \leq 2\delta$ holds, then by the extended Rips condition $d(\overline{w_1(t)}, [h]) \leq 2\delta$ and $d(\overline{hw_2(t)}, [h]) \leq 2\delta$, so again

$$d(\overline{w_1(t)}, \overline{hw_2(t)}) \leq d(\overline{w_1(t)}, [h]) + |h| + d([h], \overline{hw_2(t)}) \leq 4\delta + \varepsilon.$$

Thus if G is δ -hyperbolic and H is ε -left-bounded, and w_1 and w_2 are H -geodesic strings with $\overline{w_1a} = \overline{hw_2}$ for some $h \in H$ then for all t

$$d(\overline{w_1(t)}, \overline{hw_2(t)}) \leq 4\delta + \varepsilon.$$

We say that w_1 and w_2 have *bounded coset word difference*: the difference between them, $d(\overline{w_1(t)}, \overline{hw_2(t)}) = |\overline{w_1(t)}^{-1} \overline{hw_2(t)}|$, is at most $4\delta + \varepsilon$.

The word difference has the two essential properties of something to be calculated by a finite state automaton: it takes only finitely many values and its next value can be calculated directly from its current one if we know what caused the transition. We shall use this to construct multipliers using the standard methods for automatic groups – see [CEHLPT] for details.

Constructing Automata

Let $B_{4\delta+\varepsilon}$ be the ball of radius $4\delta + \varepsilon$ around the identity in G . We shall construct a multiplier automaton M_a with the subsets of this as its states.

We know we must start off with some $h \in H$ with $|h| \leq \varepsilon$ and end up a apart. So we let the start state be the ball of radius ε about the identity in H , and we let the accept states be all the subsets containing a .

For each pair (g_1, g_2) where $g_1, g_2 \in A'$, there is to be an arrow with this label from state S to S' where

$$S' = \overline{g_1}^{-1} S \overline{g_2} \cap B_{4\delta+\varepsilon}$$

Then for two H -geodesics w_1 and w_2 , $(w_1, w_2) \in L(M_a)$ if and only if $H\overline{w_1 a} = H\overline{w_2}$.

Similarly we can make an M_ϵ where the accept states are the subsets containing the identity. In particular, if w is an H -geodesic, $(w, w) \in L(M_\epsilon)$.

We could make an automatic system with what we now have, but the language of the word acceptor would be hard to describe – it would have to lie in the intersections of all the left and right hand sides of the multipliers and could easily contain strings of the form $a \cdot a^{-1}$. We shall instead find a way of constructing a word acceptor that just accepts a subset of this, the H -geodesic strings.

Constructing the Language of Geodesics

Now we have the M_a , we can apply the method in Theorem 3.2.2 of [CEHLPT] to get a word acceptor for the geodesics:

Let $L = \text{Left}(M_\epsilon)$. For $a \in A \cup \{\epsilon\}$, the language

$$L_a = \{w \in L : (\forall w' \in A^*)(w, w') \in L(M_a) \implies |w| < |w'|\}$$

is regular because we can do predicate calculus on regular languages, and $(|w| < |w'|)$ can be tested for by an automaton.

Similarly,

$$L' = (\cup_{a \in A} L_a a \cup \{\epsilon\}) \cap L$$

is also regular. We let L'' be the largest prefix-closed subset of L' . This is regular, since it corresponds to making every failure state a terminal failure state.

We will show that L'' coincides with the language L_0 of H -geodesic strings in L . First note that $L_0 \subseteq L$: if u is an H -geodesic, then $(u, u) \in L(M_\epsilon)$ by an earlier statement. The empty geodesic, ϵ is thus in L . So, since clearly $\epsilon \in L''$, $\epsilon \in L_0 \cap L''$.

Now take $w \in L''$ with $|w| > 0$ and suppose that, as strings, $w = ua$ for some $a \in A$. Then $u \in L''$ since L'' is prefix closed, and we may assume inductively that $u \in L_0$.

Let $v \in L_0$ with $\bar{v} = \overline{ua} = \overline{w}$. There will be such a v , since all H -geodesics are in L . Since u and v are H -geodesics, by the above construction $(u, v) \in L(M_a)$. By definition of L'' , we have $w = ua \in L'$, and so $u \in L_a$, which implies that v is strictly longer than u . But

$$d(H, H\overline{w}) = d(H, H\overline{v}) = |\overline{v}| = |v| = |u| + 1 = |w|$$

which implies that $w = ua$ is also an H -geodesic, so $w \in L_0$.

Conversely, take $w \in L_0$. We show by induction on the length of w that any prefix u of w is in L' . We may assume $u \neq \epsilon$ so, as strings, $u = va$ for some $a \in A$. Since v is also an H -geodesic, $v \in L$ by induction. We must show that $v \in L_a$, i.e. that if $(v, v') \in L(M_a)$ then $|v'| > |v|$. But u is an H -geodesic representing the same element as v' . Hence $|v'| \geq |u| = |v| + 1$.

We have now constructed L_0 , the language of H -geodesics, and shown that it is regular, so can be recognised by the word acceptor. We already have a set of multipliers which will do, though if we wanted we could create an automaton which accepts

$$L_0 \times L_0 = \{(u, v): u, v \in L_0\}$$

and intersect each of the multipliers with it.

We have thus shown that G/H is H -geodesic coset automatic, i.e. that it is coset automatic with respect to the language of H -geodesics.

Since the predicate ($u < v$ in the Short-Lex ordering) can be tested by an automaton, we can further restrict L_0 to show that G/H is coset automatic with respect to the language of Short-Lex least H -geodesics, giving us a language with a unique geodesic representative for every coset.

For, take an automaton C with

$$L(C) = \{(u, v): u > v \text{ in Short-Lex}\}$$

then set

$$L'_0 = L_0 \setminus \text{Left}(M_\epsilon \cap C)$$

and use L'_0 as the word acceptor.

So G/H is Short-Lex coset automatic, and the algorithm given in earlier chapters will find a coset automatic structure for it.

Quasiconvexity

We now show that quasiconvexity implies left-boundedness. It is not known if the converse is true.

$H \leq G$ is ε -quasiconvex if any geodesic in G between two points in H is never further than ε from H . For example, subgroups of finite index k are clearly k -quasiconvex, and it was shown in Chapter 5 that finitely-generated subgroups of free groups are as well.

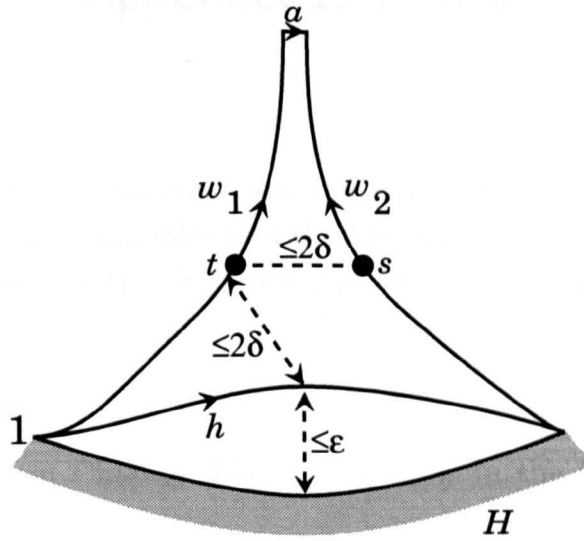


Diagram 10.5: Quasiconvexity

Suppose $H \leq G$ is ε -quasiconvex and G is δ -hyperbolic.

If w_1 and w_2 are H -geodesic strings with $H\overline{w_1 a} = H\overline{w_2}$ then $d(\overline{w_i(t)}, H) = t$. Also, $d([h], H) \leq \varepsilon$ by quasiconvexity.

Now

$$d(\overline{w_1(t)}, H) \leq d(\overline{w_1(t)}, [h]) + d([h], H).$$

So, if we assume the strings w_1 and w_2 are long enough that we can take $t = 2\delta + \varepsilon + 1$, $d(\overline{w_1(t)}, [h]) > 2\delta$.

So $d(\overline{w_1(2\delta + \varepsilon + 1)}, h[w_2]) \leq 2\delta$, by the extended Rips condition.

There is thus some s with

$$d(\overline{w_1(2\delta + \varepsilon + 1)}, h\overline{w_2(s)}) \leq 2\delta.$$

Since w_2 is an H -geodesic, $|s - (2\delta + \varepsilon + 1)| \leq 2\delta$, so $\varepsilon + 1 \leq s \leq 4\delta + \varepsilon + 1$.

Now $|h| \leq t + 2\delta + s$, so

$$|h| \leq 8\delta + 2\varepsilon + 1$$

and thus H is left-bounded.

Thus a quasiconvex subgroup of a hyperbolic group is strongly geodesically coset automatic and therefore Short-Lex coset automatic, and the algorithm given in earlier chapters will work on it. The converse is unproven.

Appendix 1: Results

Aim

This section contains several coset automata and their associated pictures, together with timing and other relevant information. All these images are available electronically from the author in PostScript or bitmap form.

Half Tetrahedron

The example given in Chapter 3 is of reflections in the faces of a hyperbolic tetrahedron.

A conjugation matrix is a matrix

$$M = \begin{pmatrix} \alpha & \beta \\ \gamma & \delta \end{pmatrix}$$

where $\alpha, \beta, \gamma, \delta \in \mathbb{C}$, which acts on $z \in \mathbb{C}$ by

$$z \mapsto \frac{\alpha \bar{z} + \beta}{\gamma \bar{z} + \delta}.$$

Let a, b, c, d be given approximately by the following conjugation matrices, determined using a program written by Oliver Goodman:

$$a = \begin{pmatrix} 0 & 1.5672969245 \\ 0.6380411933 & 0 \end{pmatrix}$$

$$b = \begin{pmatrix} -0.99051834 & 0.021117593 \\ 0.8937296094 & 0.99051834 \end{pmatrix}$$

$$c = \begin{pmatrix} 0.6570046777 - 0.8170294774i & -0.142506417 \\ 0.6960550661 & -0.6570046777 - 0.8170294774i \end{pmatrix}$$

$$d = \begin{pmatrix} 0.4550724315 + 0.9641029864i & -0.5792333486 \\ 0.2358039062 & -0.4550724315 + 0.9641029864i \end{pmatrix}$$

Let $G = \langle a, b, c, d \rangle$; then G has a presentation of the form

$$G = \langle a, b, c, d \mid a^2, b^2, c^2, d^2, (ab)^4, (ac)^3, (ad)^2, (bc)^4, (bd)^4, (cd)^4 \rangle$$

Let C be the circle of centre $-1.0975351456 - 2.0161336273i$ and radius 1.6771876905, and let $H = \text{Stab}_G(C)$. Then H is given by

$$H = \langle a, b, c \rangle.$$

The word acceptor for G/H takes 5 seconds to make, and is:

	a	b	c	d		a	b	c	d		a	b	c	d			
1	A	2	2	2	3;	2	N	2	2	2	2;	3	A	2	4	5	2;
4	A	6	2	7	8;	5	A	9	10	2	11;	6	A	2	12	7	13;
7	A	14	10	2	3;	8	A	2	2	5	2;	9	A	2	15	2	16;
10	A	17	2	18	3;	11	A	2	19	2	2;	12	A	2	2	7	3;
13	A	2	20	5	2;	14	A	2	15	2	3;	15	A	17	2	21	3;
16	A	2	4	22	2;	17	A	2	12	7	3;	18	A	23	2	2	3;
19	A	6	2	21	8;	20	A	24	2	7	8;	21	A	14	25	2	3;
22	A	2	10	2	11;	23	A	2	26	2	3;	24	A	2	2	7	13;
25	A	27	2	2	3;	26	A	12	2	21	3;	27	A	2	12	28	3;
28	A	2	10	2	3;												

By using depth-first search on this word acceptor (see Chapter 4) we can produce the picture on the next page. It was run with a minimum radius of 10^{-5} and a visibility radius of 10^{-3} . It examined 190,833,325 circles, of which 65,236 were visible; the maximum word length reached was 722, although no circles of word length more than 62 were visible. For definitions of these terms, see page 26.

The whole process took Crocus, a Sun SparcServer 2000 belonging to University of Warwick Computer Services, 6 hours and 54 minutes of run-time; this machine, or Lupin, an equivalently powerful machine, will be used for all timings given.

The circles are coloured by word length, with the nearest circles red (including C , which is the large red central circle) and the furthest circles violet. Formally, saturation and brightness are always 1 and hue is $depth/max_depth$, where $depth$ is the word length of the current circle and max_depth is the longest word length for a visible circle: in this case, 62.

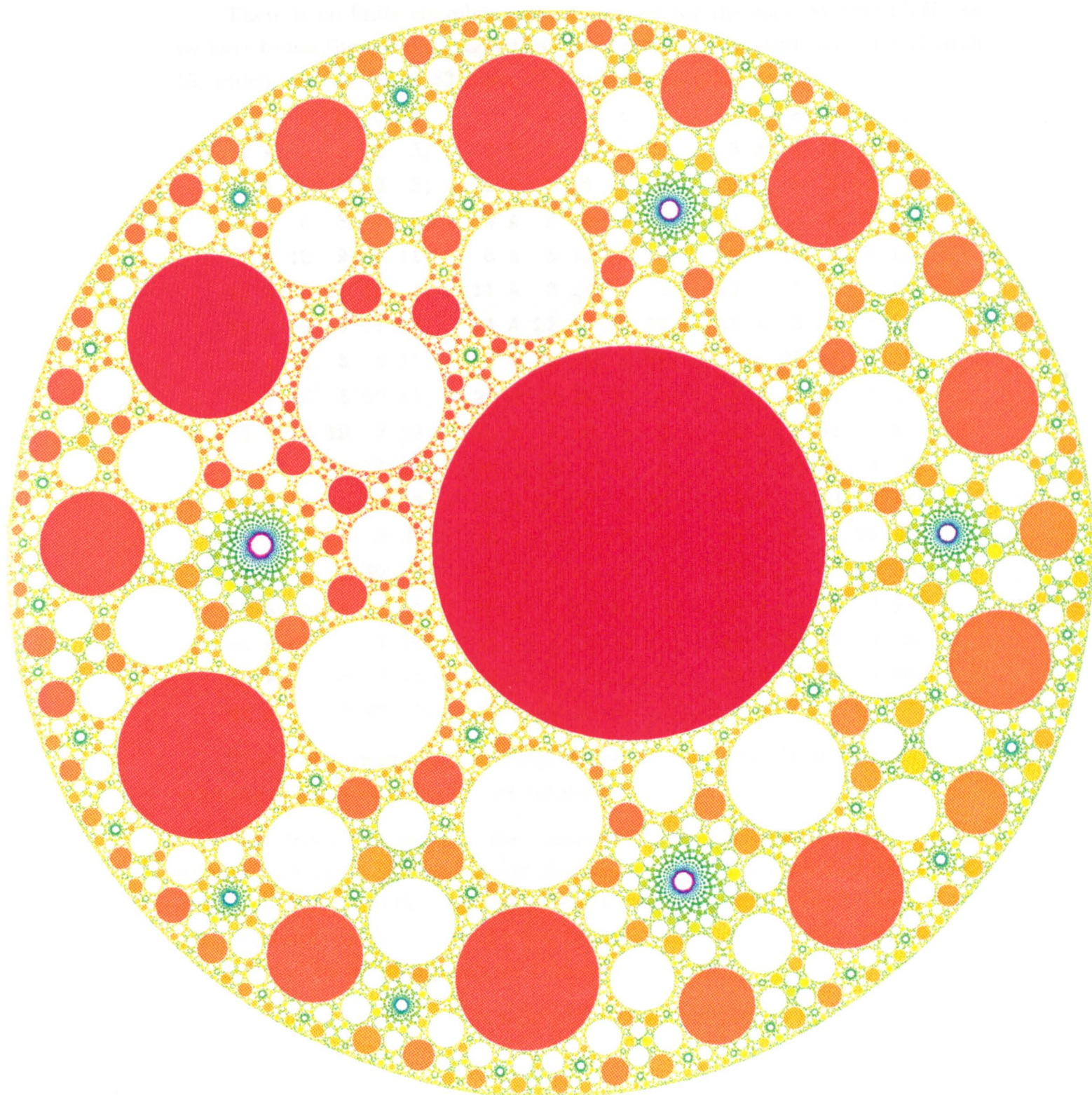
Note that matrices multiply on the left but our words multiply on the right. Note also that conjugation matrices do not multiply like ordinary matrices because of the conjugation involved.

Full Tetrahedron

The picture on the left shows a full tetrahedron. The blue circle in the center is the largest sphere that can be packed inside the tetrahedron. The spheres in the next layer are the largest spheres that can be packed around the central sphere. The spheres in the next layer are the largest spheres that can be packed around the spheres in the previous layer. The spheres in the next layer are the largest spheres that can be packed around the spheres in the previous layer.

The spheres in the next layer are the largest spheres that can be packed around the spheres in the previous layer. The spheres in the next layer are the largest spheres that can be packed around the spheres in the previous layer. The spheres in the next layer are the largest spheres that can be packed around the spheres in the previous layer. The spheres in the next layer are the largest spheres that can be packed around the spheres in the previous layer.

There is no finite number of spheres that can be packed inside a tetrahedron. The number of spheres that can be packed inside a tetrahedron is infinite. The number of spheres that can be packed inside a tetrahedron is infinite. The number of spheres that can be packed inside a tetrahedron is infinite.



Full Tetrahedron

The previous picture is only half of a circle packing. The other half is the orbit of C' , the circle of centre $0.1607775601 + 0.1838178602i$ and radius 0.6264092653 . If we let $H' = \text{Stab}_G(C')$, we find that H' is given by

$$H' = \langle b, c, d \rangle.$$

There is no finite complete rewrite rule set for the coset system G/H' , so we have to use the guessing algorithm. This requires left-hand sides up to length 15, which is a rule set of 83 rules. The automaton produced is:

	a	b	c	d		a	b	c	d		a	b	c	d
1 A	2	2	2	3;	2 N	2	2	2	2;	3 A	2	4	5	2;
1 A	2	3	3	3;	2 A	3	4	5	3;	3 N	3	3	3	3;
4 A	6	3	7	8;	5 A	3	9	3	10;	6 A	3	3	7	11;
7 A	12	9	3	11;	8 A	3	13	14	3;	9 A	15	3	16	11;
10 A	3	17	18	3;	11 A	3	17	14	3;	12 A	3	19	3	11;
13 A	20	3	21	3;	14 A	22	9	3	23;	15 A	3	6	7	11;
16 A	24	3	3	11;	17 A	25	3	7	26;	18 A	27	28	3	3;
19 A	15	3	29	11;	20 A	3	30	21	3;	21 A	31	9	3	10;
22 A	3	19	3	32;	23 A	3	33	3	3;	24 A	3	34	3	11;
25 A	3	6	7	35;	26 A	3	3	14	3;	27 A	3	36	3	3;
28 A	15	3	16	8;	29 A	12	37	3	11;	30 A	3	3	7	8;
31 A	3	19	3	38;	32 A	3	17	39	3;	33 A	25	3	29	26;
34 A	6	3	29	11;	35 A	3	40	14	3;	36 A	15	3	29	8;
37 A	41	3	3	11;	38 A	3	17	42	3;	39 A	3	9	3	23;
40 A	43	3	7	26;	41 A	3	6	44	11;	42 A	45	9	3	23;
43 A	3	3	7	35;	44 A	3	9	3	11;	45 A	3	19	3	46;
46 A	3	17	47	3;	47 A	3	28	3	3;					

This automaton has maximum depth 11 and distinguishability 7. The guessing process takes Lupin 118 seconds.

The depth-first search coset enumeration algorithm was applied with a minimum radius of 10^{-5} and a visibility radius of 10^{-3} . It examined 281,484,964 circles, of which 101,005 were visible; the maximum word length reached was 865, although no circles of word length more than 69 were visible. The enumeration took 8 hours and 54 minutes to complete.

The picture shown on the next page is the amalgamation of the two halves of the circle packing.

Peppertree Forest

As described in Chapter 8, we can construct a fractal by iteratively adding the maximal disks to the disk complement of a given disk. The resulting fractal is a random fractal, and its boundary is a random fractal curve.

Figure 10.1 shows a fractal curve constructed by this process.

Figure 10.2 shows a fractal curve constructed by this process.

Figure 10.3 shows a fractal curve constructed by this process.

Figure 10.4 shows a fractal curve constructed by this process.

Figure 10.5 shows a fractal curve constructed by this process.

Figure 10.6 shows a fractal curve constructed by this process.

Figure 10.7 shows a fractal curve constructed by this process.

Figure 10.8 shows a fractal curve constructed by this process.

Figure 10.9 shows a fractal curve constructed by this process.

Figure 10.10 shows a fractal curve constructed by this process.

Figure 10.11 shows a fractal curve constructed by this process.

Figure 10.12 shows a fractal curve constructed by this process.

Figure 10.13 shows a fractal curve constructed by this process.

Figure 10.14 shows a fractal curve constructed by this process.

Figure 10.15 shows a fractal curve constructed by this process.

Figure 10.16 shows a fractal curve constructed by this process.

Figure 10.17 shows a fractal curve constructed by this process.

Figure 10.18 shows a fractal curve constructed by this process.

Figure 10.19 shows a fractal curve constructed by this process.

Figure 10.20 shows a fractal curve constructed by this process.

Figure 10.21 shows a fractal curve constructed by this process.

Figure 10.22 shows a fractal curve constructed by this process.

Figure 10.23 shows a fractal curve constructed by this process.

Figure 10.24 shows a fractal curve constructed by this process.

Figure 10.25 shows a fractal curve constructed by this process.

Figure 10.26 shows a fractal curve constructed by this process.

Figure 10.27 shows a fractal curve constructed by this process.

Figure 10.28 shows a fractal curve constructed by this process.

Figure 10.29 shows a fractal curve constructed by this process.

Figure 10.30 shows a fractal curve constructed by this process.

Figure 10.31 shows a fractal curve constructed by this process.

Figure 10.32 shows a fractal curve constructed by this process.

Figure 10.33 shows a fractal curve constructed by this process.

Figure 10.34 shows a fractal curve constructed by this process.

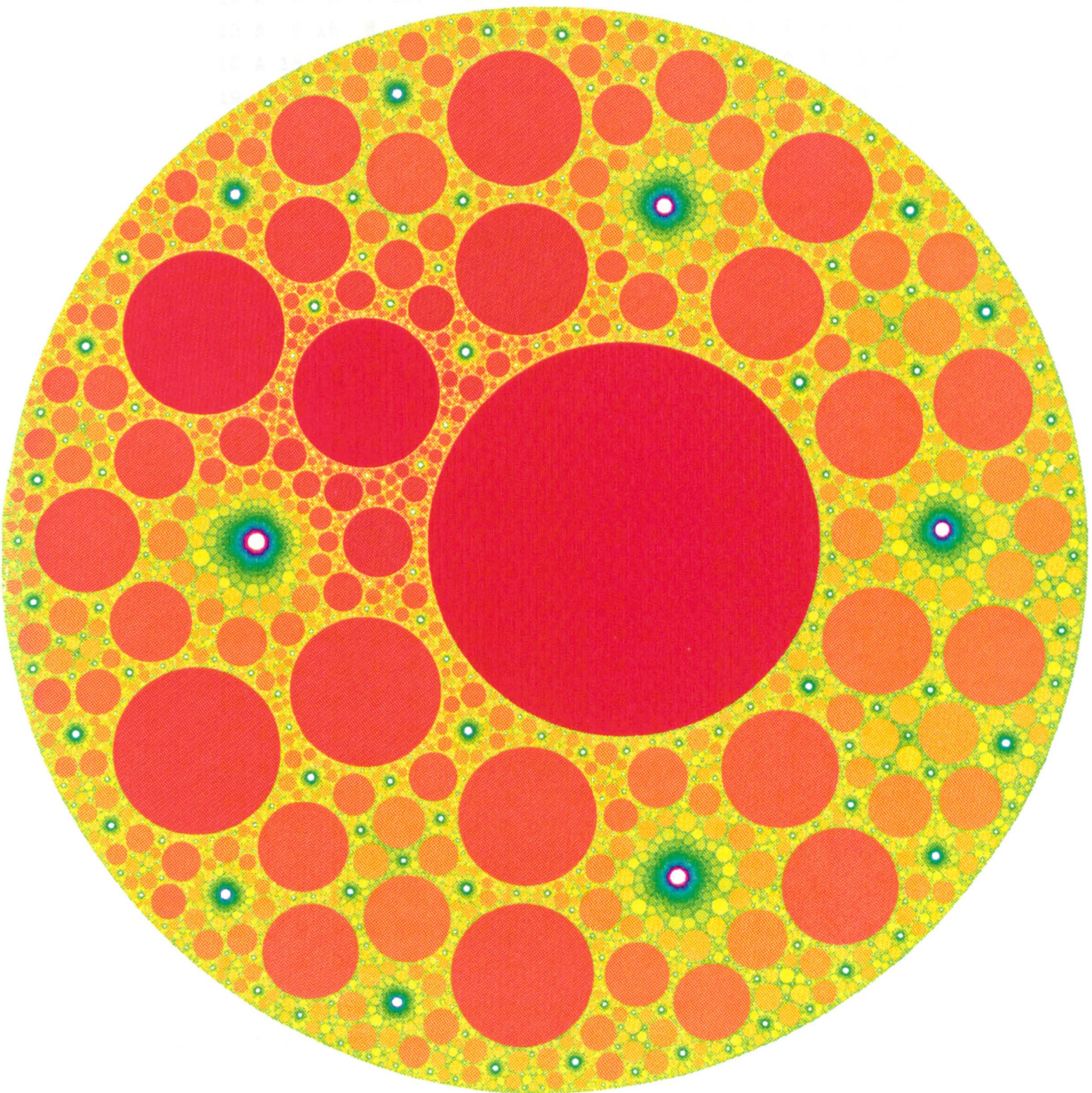
Figure 10.35 shows a fractal curve constructed by this process.

Figure 10.36 shows a fractal curve constructed by this process.

Figure 10.37 shows a fractal curve constructed by this process.

Figure 10.38 shows a fractal curve constructed by this process.

Figure 10.39 shows a fractal curve constructed by this process.



Punctured Torus

As described in Chapter 5, we can enumerate points in the Teichmüller space of the punctured torus. The first example is for $p/q = 1/11$. Here, $w = Stttttttttt$, and its complementary generator $u = Stttttttttt$. The coset word acceptor is:

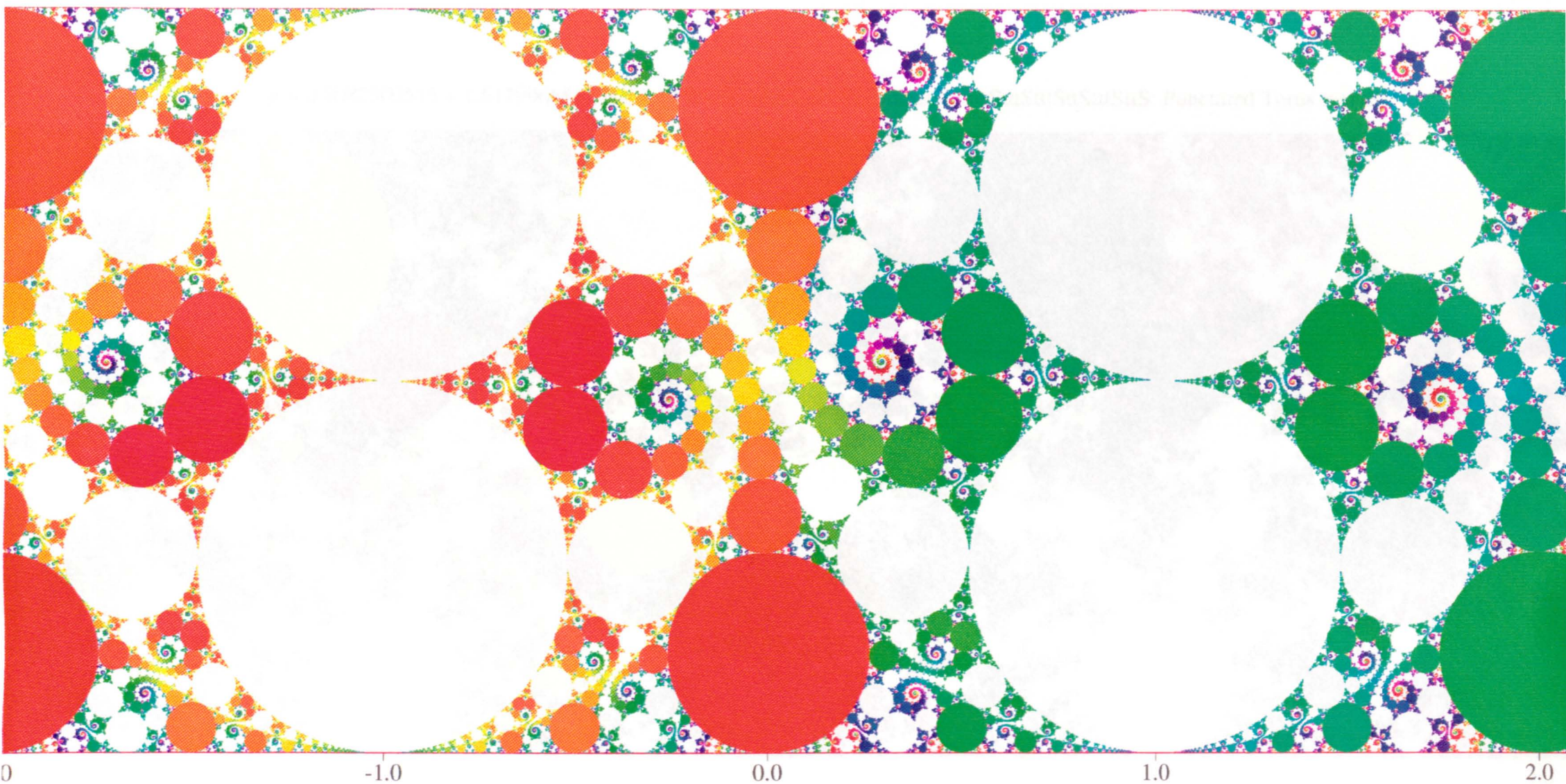
w	W	u	U	w	W	u	U	w	W	u	U
1 A	2	3	4 5;	2 A	6	7	8 9;	3 A	7	3	8 10;
4 A	6	11	8 7;	5 A	7	3	7 10;	6 A	6	7	8 10;
7 N	7	7	7 7;	8 A	6	3	8 7;	9 A	6	7	7 12;
10 A	6	3	7 10;	11 A	7	3	13 10;	12 A	14	3	7 10;
13 A	6	15	8 7;	14 A	6	7	8 16;	15 A	7	3	17 10;
16 A	18	3	7 10;	17 A	6	19	8 7;	18 A	6	7	8 20;
19 A	7	3	21 10;	20 A	22	3	7 10;	21 A	6	23	8 7;
22 A	6	7	8 24;	23 A	7	3	25 10;	24 A	26	3	7 10;
25 A	6	5	8 7;	26 A	6	7	8 7;				

The second is for $p/q = 13/34$. Here, $u = SttStttSttStttStttSttStttSttt$ and $w = SttStttSttStttStttSttStttSttStttStttStttSttt$, and the coset word acceptor is:

w	W	u	U	w	W	u	U	w	W	u	U
1 A	2	3	4 5;	2 A	2	6	7 8;	3 A	6	9	10 8;
4 A	2	6	7 6;	5 A	11	9	6 8;	6 N	6	6	6 6;
7 A	2	9	7 6;	8 A	2	9	6 8;	9 A	6	9	7 8;
10 A	6	9	12 6;	11 A	2	6	7 13;	12 A	2	14	7 6;
13 A	2	9	6 15;	14 A	6	9	16 8;	15 A	17	9	6 8;
16 A	2	9	18 6;	17 A	2	6	7 19;	18 A	2	20	7 6;
19 A	21	9	6 8;	20 A	6	9	22 8;	21 A	2	6	7 23;
22 A	2	24	7 6;	23 A	2	9	6 25;	24 A	6	9	26 8;
25 A	27	9	6 8;	26 A	2	9	28 6;	27 A	2	6	7 29;
28 A	2	30	7 6;	29 A	2	9	6 31;	30 A	6	9	32 8;
31 A	33	9	6 8;	32 A	2	9	34 6;	33 A	2	6	7 35;
34 A	2	36	7 6;	35 A	37	9	6 8;	36 A	6	9	38 8;
37 A	2	6	7 39;	38 A	2	40	7 6;	39 A	2	9	6 41;
40 A	6	9	42 8;	41 A	43	9	6 8;	42 A	2	9	44 6;
43 A	2	6	7 45;	44 A	2	46	7 6;	45 A	47	9	6 8;
46 A	6	9	48 8;	47 A	2	6	7 49;	48 A	2	50	7 6;
49 A	2	9	6 51;	50 A	6	9	52 8;	51 A	53	9	6 8;
52 A	2	9	54 6;	53 A	2	6	7 55;	54 A	2	56	7 6;
55 A	2	9	6 57;	56 A	6	9	58 8;	57 A	59	9	6 8;
58 A	2	9	4 6;	59 A	2	6	7 60;	60 A	6	9	6 8;

The picture shown is coloured in depth bands: depth 0–10 is red; 10–20 yellow; 20–40 green; 40–80 cyan; 80–160 blue and 160–723 magenta.

$\mu = 0.02768058389 + 1.926780321i$; $p/q = 1/11$; $w = \text{tttttttttS}$: Punctured Torus real set



Visibility Radius = 0.0005

Minimum Radius = $1e-07$

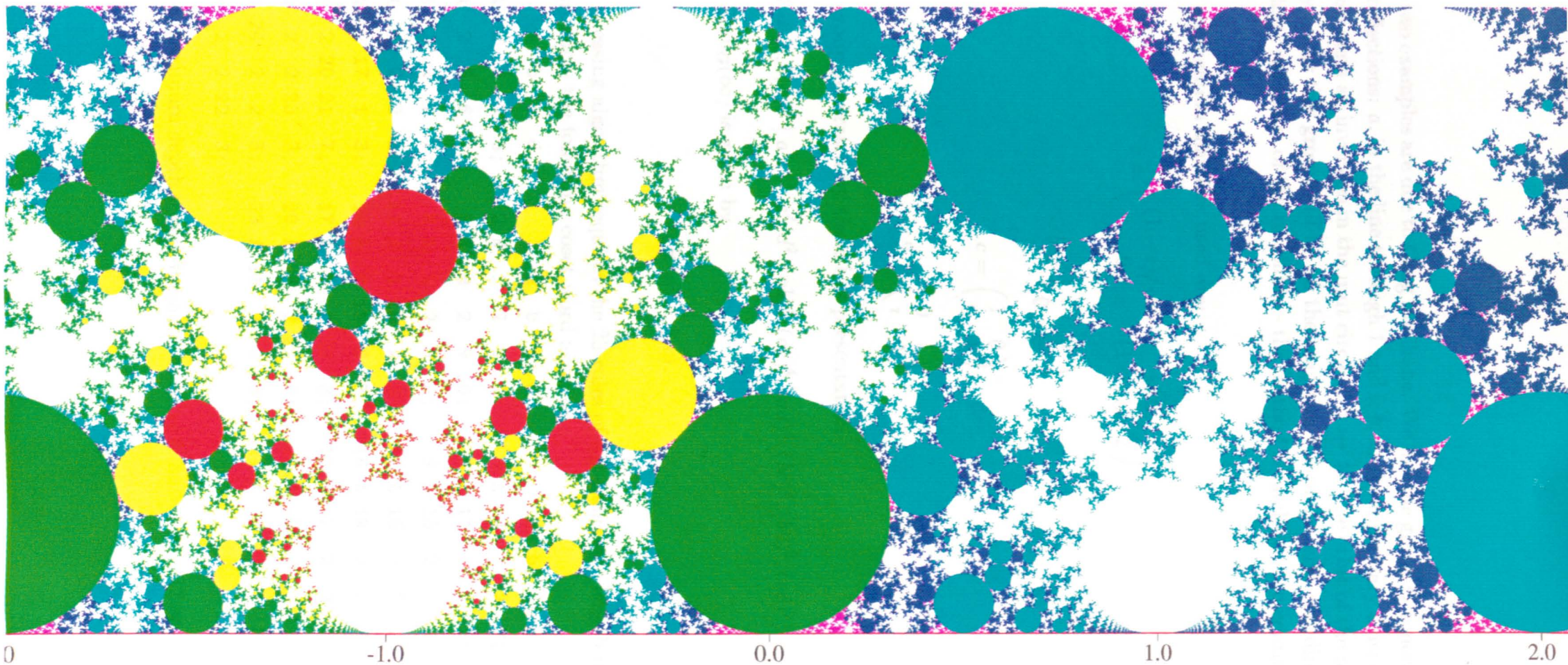
Run by marab@crocus.csv.warwick.ac.uk on Sun Sep 12 12:43:37 1993

Maximum depth: 258/1020

Number of circles: 51286/139391631

Runtime: 4 hrs 36 mins 21 secs

$\mu = 0.7082909515 + 1.617996654i$; $p/q = 13/34$; $w = \text{tttStttSttStttSttSttSttSttSttSttSttSttS}$: Punctured Torus real set



Visibility Radius = 0.0005

Minimum Radius = $1e-10$

Run by marab@crocus.csv.warwick.ac.uk on Fri Aug 27 22:23:26 1993

Maximum depth: 723/2615

Number of circles: 100723/2119352826

Runtime: 3 days 6 hrs 3 mins

Square

The next two examples are due to Greg McShane. We use the group G generated by three reflections: a , in the line through 1 and i ; b , in the imaginary axis; c , in the real axis; and an inversion in the unit circle d : $z \mapsto \frac{1}{\bar{z}}$. The orbit of the point at infinity under this group will be all the points in the plane with coordinates in some finite extension of the rationals – the method is analogous to continued fractions.

The conjugation matrices used are:

$$a = \begin{pmatrix} (1-i)/\sqrt{2} & \sqrt{2}i \\ 0 & (1+i)/\sqrt{2} \end{pmatrix}$$

$$b = \begin{pmatrix} i & 0 \\ 0 & -i \end{pmatrix}$$

$$c = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$$

$$d = \begin{pmatrix} 0 & i \\ i & 0 \end{pmatrix}$$

The group generated by these has presentation

$$G = \langle a, b, c, d \mid a^2, b^2, c^2, d^2, (ab)^4, (ac)^4, (ad)^4, (bc)^2, (bd)^2, (cd)^2 \rangle$$

and $H = \text{Stab}_G(\infty)$ is given by

$$H = \langle a, b, c \rangle.$$

The guessing algorithm requires the 23 rules of left-hand side length up to 13, and produces the following coset word acceptor:

	a	b	c	d		a	b	c	d		a	b	c	d
1 A	2	2	2	3;	2 N	2	2	2	2;	3 A	4	2	2	2;
4 A	2	5	6	7;	5 A	8	2	9	10;	6 A	11	2	2	12;
7 A	2	2	2	2;	8 A	2	2	9	3;	9 A	13	2	2	3;
10 A	14	2	2	2;	11 A	2	15	2	3;	12 A	16	2	2	2;
13 A	2	17	1	3;	14 A	2	18	6	7;	15 A	19	2	2	3;
16 A	2	20	21	7;	17 A	8	2	22	3;	18 A	2	2	9	10;
19 A	2	2	23	3;	20 A	8	2	22	10;	21 A	2	2	2	12;
22 A	24	2	2	3;	23 A	11	2	2	3;	24 A	2	25	1	3;
25 A	2	2	22	3;										

It has maximum depth 9 and distinguishability 4, and takes 112 seconds to generate.

Colouring this and subsequent pictures by word length does not produce anything interesting: because of the coarseness of the grid, most of the areas that do get coloured get overwritten maybe a thousand times, and the result is a page of coloured dust, with the colour of the current depth of the search predominating. Note also that minimum radius techniques clearly do not work with points, and that almost no images of infinity lie outside $|z| \leq 10$ up to words of length 100.

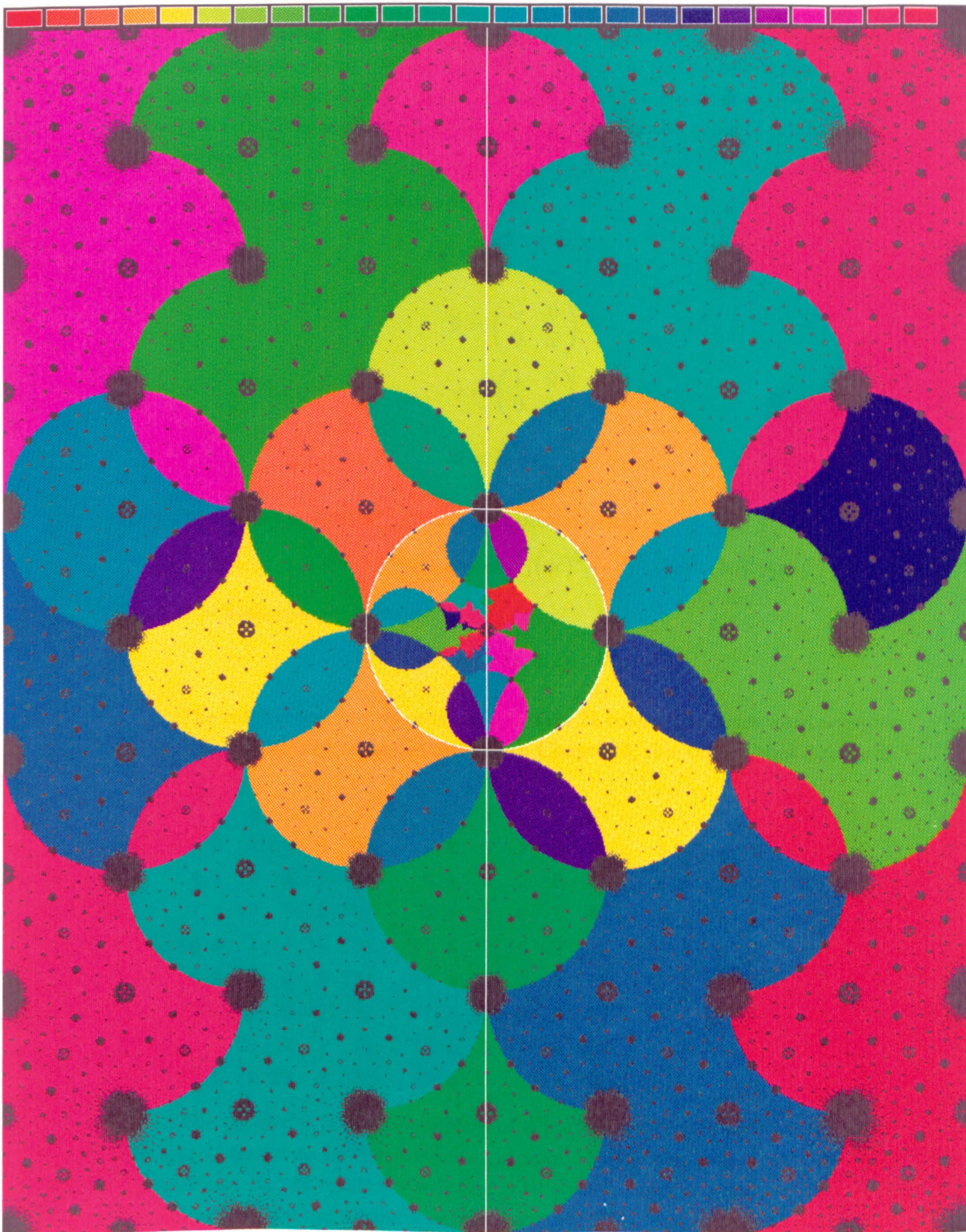
The only interesting way to colour the points plotted, then, is by the state the word acceptor is in at the time. To my surprise, this gives a coherent picture. The picture shown is a partial search to length 80, run for 24 hours at a rate of approximately a million words per minute. The unit circle is marked, and the picture boundaries are $\pm 3 \pm 2i$.

Down the right-hand side of the picture is a table of colours by state, with the start state, 1, at the top. All of the states give rise to coloured regions, with the exception, of course, of state 2, which is a fail state. Every region corresponding to a particular state appears to be connected. The region corresponding to the start state is in the bottom right-hand corner in red.

The second picture is drawn with exactly the same parameters as the first, but instead of being coloured by the current state, the points are coloured by the state the word acceptor was in one letter previously. This gives a 'Markov partition' of the regions into finitely many subregions. Note that, because the generators have order 2, all the regions must change colour.

Not all of them need split, though: states 4, 5, 13–21 and 23–25 are only reached from a single state, so their regions will not split. All other regions split into at most three pieces, with the exception of the central region: this corresponds to state 3 and is broken into 12 subregions, which are the images under d of the outside of the picture. We must also lose a colour, since state 7 is a terminal accept state.





Hexagon

The second of Greg McShane's examples is also generated by three reflections and an inversion: a , in the line through 1 and ω ; b , in the line through 0 and ω ; c , in the real axis; and an inversion in the unit circle $d: z \mapsto \frac{1}{\bar{z}}$. Here, $\omega = (1 + i\sqrt{3})/2$ is a cube root of -1 . Again, we take the orbit of infinity under this group.

The conjugation matrices used are:

$$a = \begin{pmatrix} \bar{\omega} & \sqrt{3}i \\ 0 & \omega \end{pmatrix}$$

$$b = \begin{pmatrix} \omega & 0 \\ 0 & \bar{\omega} \end{pmatrix}$$

$$c = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$$

$$d = \begin{pmatrix} 0 & i \\ i & 0 \end{pmatrix}$$

The group generated by these has presentation

$$G = \langle a, b, c, d \mid a^2, b^2, c^2, d^2, (ab)^3, (ac)^3, (ad)^6, (bc)^3, (bd)^2, (cd)^2 \rangle$$

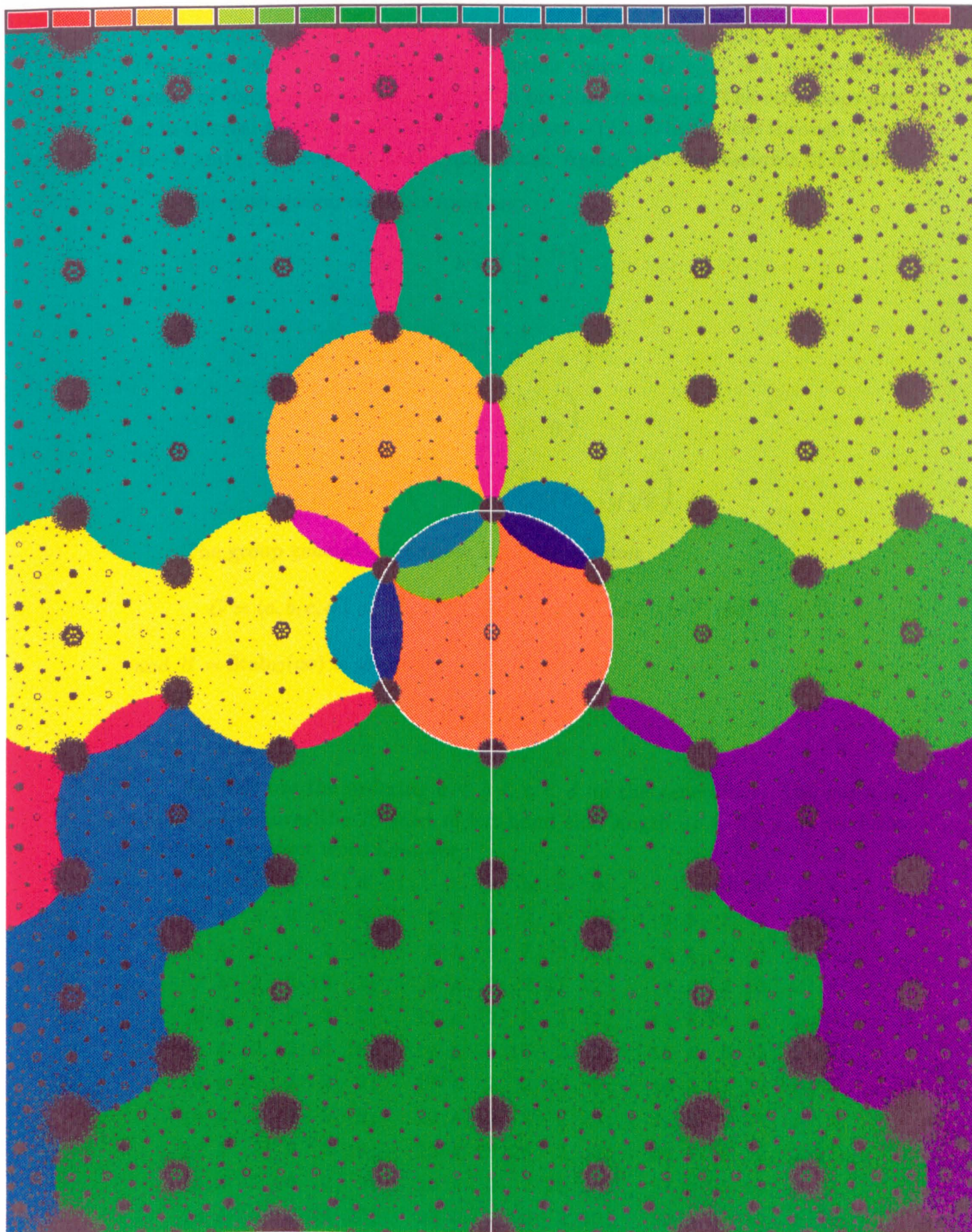
and $H = \text{Stab}_G(\infty)$ is given by

$$H = \langle a, b, c \rangle.$$

The rule system is confluent and contains 22 rules. The coset word acceptor is therefore produced using the method in Chapter 3; this takes about 5 seconds. It is:

	a	b	c	d		a	b	c	d		a	b	c	d
1	A	2	2	3;	2	N	2	2	2;	3	A	4	2	2;
4	A	2	5	7;	5	A	2	2	8;	6	A	2	9	3;
7	A	10	2	2;	8	A	11	9	2;	9	A	12	2	2;
10	A	2	13	15;	11	A	2	16	2;	12	A	2	2	6;
13	A	2	2	8;	14	A	2	9	2;	15	A	2	2	2;
16	A	5	2	19;	17	A	20	2	2;	18	A	21	2	2;
19	A	22	2	2;	20	A	2	2	6;	21	A	2	23	2;
22	A	2	23	2;	23	A	2	2	19;					

The picture is a state colouring of a partial search to depth 80 for about 24 hours, drawn as in the previous example. A full search to depth 40 would cover about 52,233 million words and take about a month; good results can be obtained from partial deeper searches, however. Note that one of the red regions is disconnected.



This example was suggested by André Rocha, and is found in [FK], and in [Mag] (but there is a mistake in the presentation there). Here, G is a Klein–Fricke group of conjugation matrices, as in the previous examples. The reflections are: a , in the real axis; b , in the line with real part $-1/2$; d , in the line through 0 and $1 - i$; and c , an inversion in the unit circle. Again, we take the orbit of infinity.

$$a = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$$

$$b = \begin{pmatrix} -i & -i \\ 0 & i \end{pmatrix}$$

$$c = \begin{pmatrix} 0 & i \\ i & 0 \end{pmatrix}$$

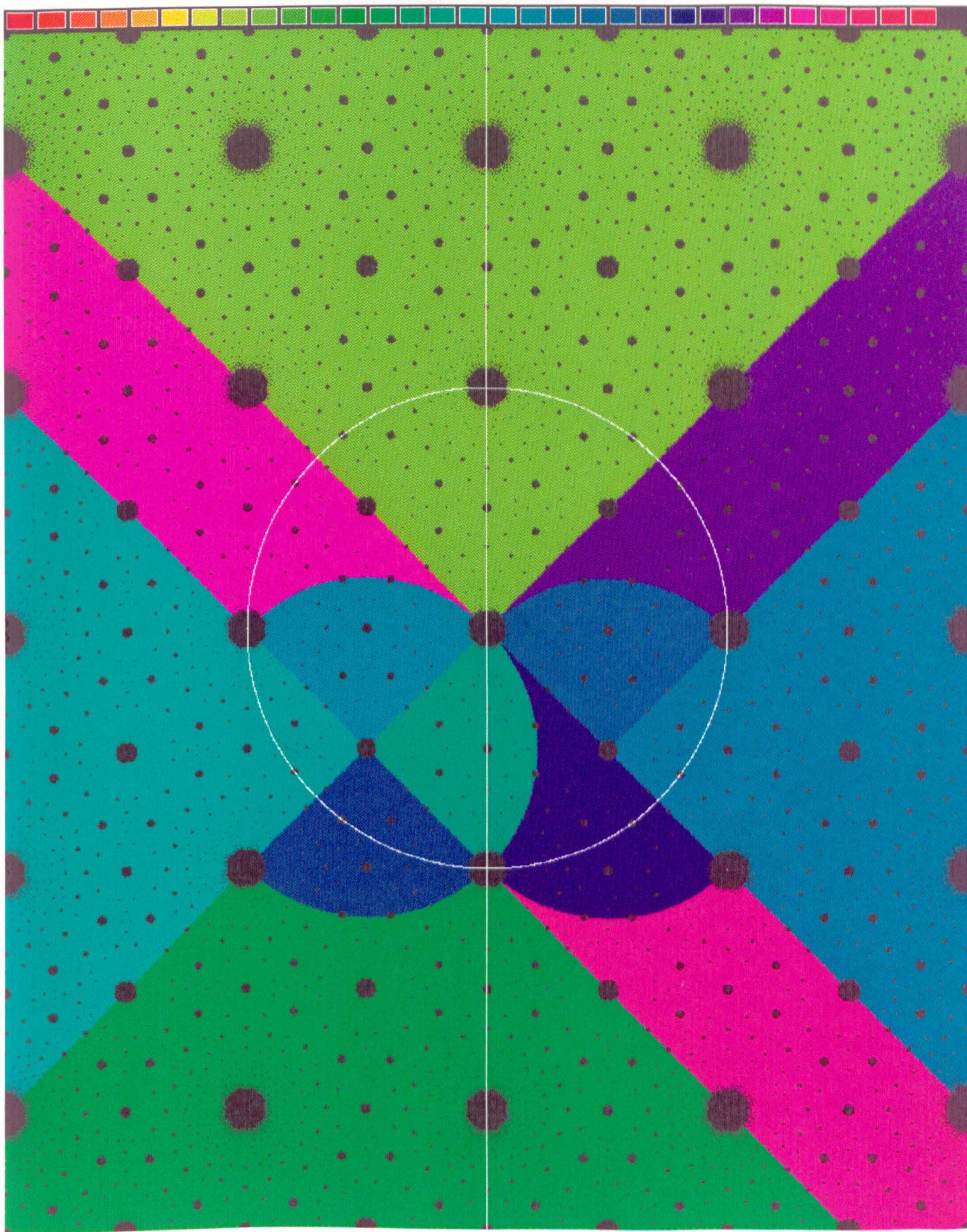
$$d = \begin{pmatrix} (1-i)/\sqrt{2} & 0 \\ 0 & (1+i)/\sqrt{2} \end{pmatrix}$$

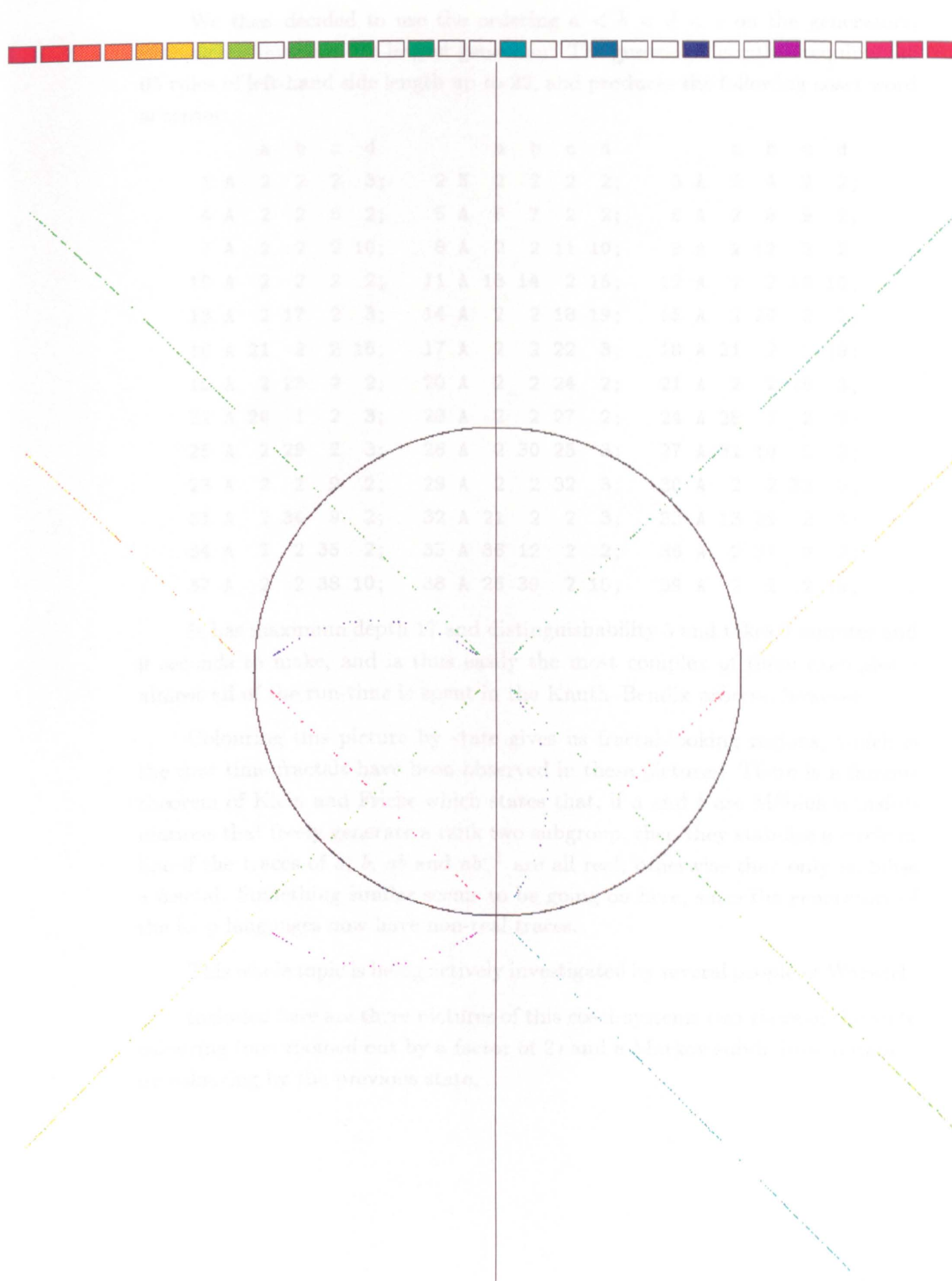
$$G = \langle a, b, c, d \mid a^2, b^2, c^2, d^2, (ab)^2, (ac)^2, (ad)^4, (bc)^3, (bd)^4, (cd)^2 \rangle$$
$$H = \langle a, b, d \rangle.$$
[illegible]

It has maximum depth 14 and distinguishability 6, and takes 31 seconds to make.

The word acceptor splits into essentially two strongly connected parts: states 5, 6, 8, 11, 12, 14, 17, 20, 23, 26, 29, 30 and 31 form one loop, and states 9, 13, 15, 16, 18, 19, 21, 22, 24, 27 and 28 form the other. Of the other states, 1, 3 and 4 are transient; 2 is the terminal fail state; and 7 and 10 are on a siding from the first loop. It is possible to get from the first loop to the second, but not back again.

This separation of the states into two sets is mirrored in the picture. The first set of states corresponds to lines and arcs; the second corresponds to regions – these are the ‘thin’ and ‘fat’ states respectively. Examining the language of loops at a thin state, such as 14, we find it to be a finitely generated free semigroup; it has three free generators: *bdabda*, *bdabcda* and *bcdabcdabda*. All of the fat states, however, have infinitely generated loop languages because there are two disjoint loops (e.g. 27, 25, 28 and 9, 13, 16, 19) in that half of the partition. The second picture shows only the thin states – state 14 corresponds to the line through 0 and $1 - i$.





We then decided to use the ordering $a < b < d < c$ on the generators, making the inversion the largest generator. The guessing algorithm requires the 65 rules of left-hand side length up to 22, and produces the following coset word acceptor:

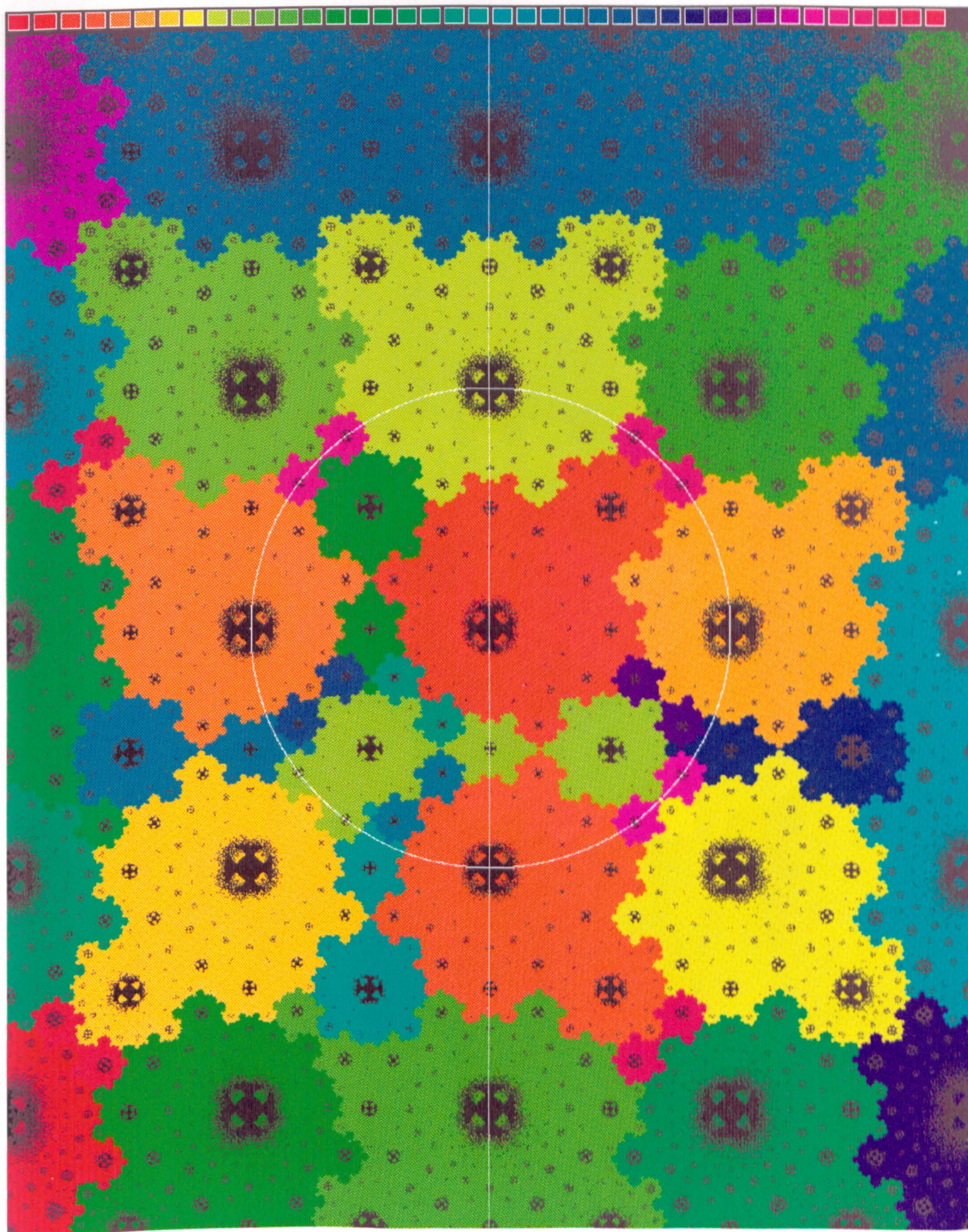
	a	b	c	d		a	b	c	d		a	b	c	d			
1	A	2	2	2	3;	2	N	2	2	2	2;	3	A	2	4	2	2;
4	A	2	2	5	2;	5	A	6	7	2	2;	6	A	2	8	9	2;
7	A	2	2	2	10;	8	A	2	2	11	10;	9	A	2	12	2	2;
10	A	2	2	2	2;	11	A	13	14	2	15;	12	A	2	2	16	10;
13	A	2	17	2	3;	14	A	2	2	18	19;	15	A	2	20	2	2;
16	A	21	2	2	15;	17	A	2	2	22	3;	18	A	21	2	2	19;
19	A	2	23	2	2;	20	A	2	2	24	2;	21	A	2	2	25	3;
22	A	26	1	2	3;	23	A	2	2	27	2;	24	A	28	2	2	2;
25	A	2	29	2	3;	26	A	2	30	25	3;	27	A	31	10	2	2;
28	A	2	2	9	2;	29	A	2	2	32	3;	30	A	2	2	33	3;
31	A	2	34	9	2;	32	A	21	2	2	3;	33	A	13	29	2	3;
34	A	2	2	35	2;	35	A	36	12	2	2;	36	A	2	37	2	2;
37	A	2	2	38	10;	38	A	26	39	2	15;	39	A	2	2	2	19;

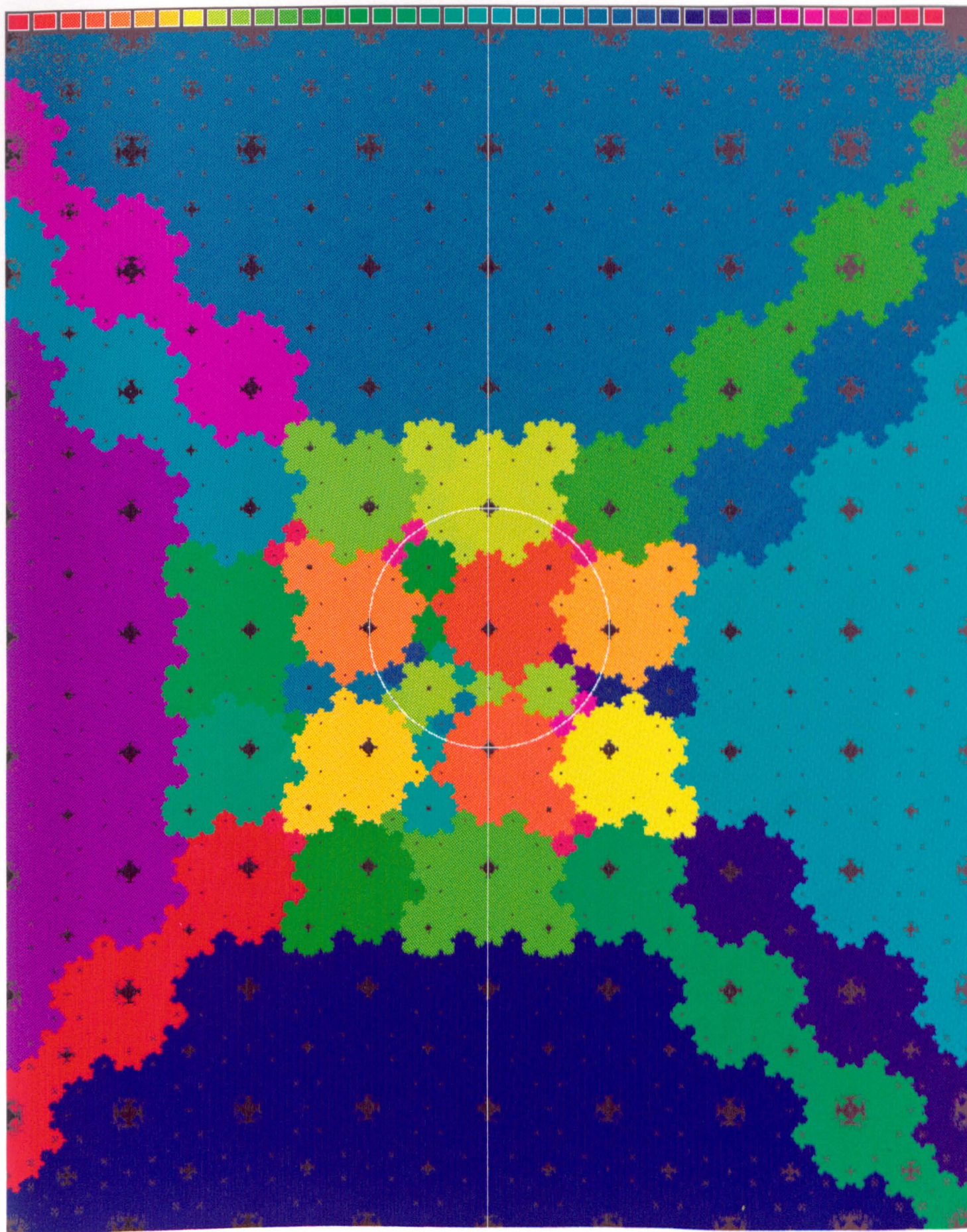
It has maximum depth 17 and distinguishability 5 and takes 8 minutes and 9 seconds to make, and is thus easily the most complex of these examples – almost all of the run-time is spent in the Knuth–Bendix process, however.

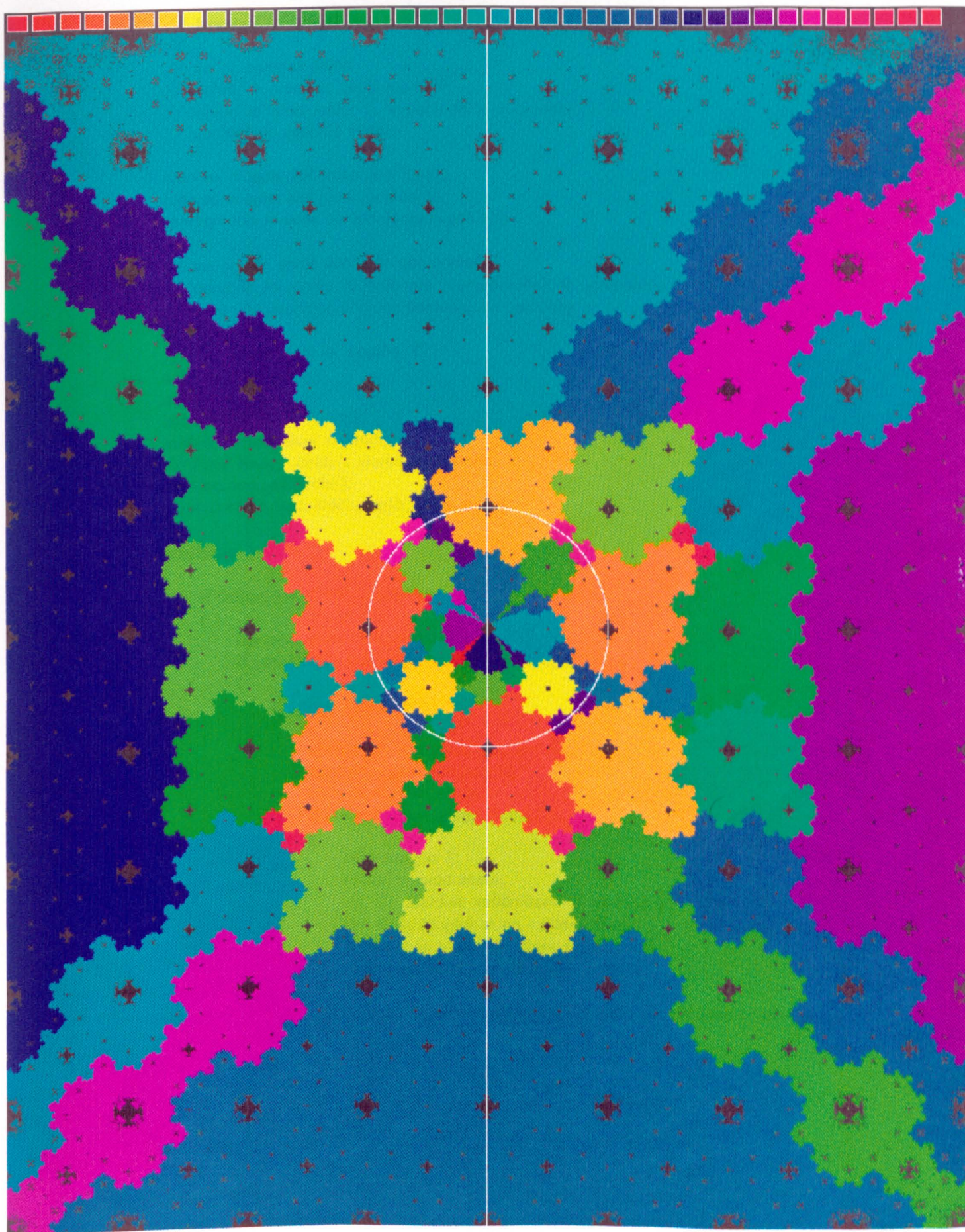
Colouring this picture by state gives us fractal-looking regions, which is the first time fractals have been observed in these pictures. There is a famous theorem of Klein and Fricke which states that, if a and b are Möbius transformations that freely generate a rank two subgroup, then they stabilise a circle or line if the traces of a , b , ab and ab^{-1} are all real; otherwise they only stabilise a fractal. Something similar seems to be going on here, since the generators of the loop languages now have non-real traces.

This whole topic is being actively investigated by several people at Warwick.

Included here are three pictures of this coset system: two views of the state colouring (one zoomed out by a factor of 2) and a Markov subdivision obtained by colouring by the previous state.







Appendix 2: Programs

```
// Example 1: Depth-first search
//
// nn = current depth
// ns[nn] = current state number
// nl[nn] = next letter to explore from this state
// nm[nn] = current matrix

int* ns = new int [max_depth+1];
int* nl = new int [max_depth+1];
matrix* nm = new matrix [max_depth+1];

// wa is the word acceptor automaton
// mlist is an array of matrices corresponding to generators
// m is the starting matrix; ususally the identity
nm[0] = m;
// The start state is usually 1
ns[0] = start_state;
nl[0] = 1;
int nn = 0;
matrix current_mat = nm[0];
// Find image circle's centre and radius
current_mat.set_circle();
print(current_mat, start_depth, start_state);

while(nn ≥ 0)
{
    if (nl[nn]++ ≤ wa.num_letters())
    {
        int next_state = wa[ns[nn]] [nl[nn]-1];
        if (next_state && wa[next_state].is_accept())
        {
            if (nn < max_depth)
            {
                nm[nn+1] = mlist[nl[nn]-2] * nm[nn];
                current_mat = nm[nn+1];
                current_mat.set_circle();
                if (current_mat.is_significant())
                {
                    nl[++nn] = 1;
                    ns[nn] = next_state;
                    // Do Markov subdivision if necessary
                    int state_to_plot = nn - state_offset;
                    if (state_to_plot < 0) state_to_plot = 0;
                    if (current_mat.is_visible())
                    {
                        print(current_mat, start_depth+nn, ns[state_to_plot]);
                    }
                }
            }
        }
        else
        {
            nn--;
        }
    }
}
```

```

// Example 2: calculate length of first word separating the tail languages
//
int differ_depth(const automaton& aut, int s1, int s2, table& first_differ)
{
    // Do we know the answer already?
    if (s1 ≤ first_differ.height() && s2 ≤ first_differ.height())
    {
        if (first_differ.get(s1, s2) ≠ -1)
        {
            return first_differ.get(s1, s2);
        }
    }
    else
    {
        // Clear space in table
        int old_height = first_differ.height();
        int larger = s1;
        if (s2 > s1) larger = s2;
        first_differ.set(larger, aut.num_states(), -1);
        int i, j;
        for (i=old_height+1; i≤larger; i++)
        {
            for (j=1; j≤first_differ.width(); j++)
            {
                first_differ.set(i, j, -1);
            }
        }
    }
}

int differ_at = 0;

// Make sure we don't pass this way again or we could easily get an infinite loop
first_differ.set(s1, s2, aut.num_states()+1);
first_differ.set(s2, s1, aut.num_states()+1);

// Do we differ on the empty word?
if (aut[s1].is_accept() == aut[s2].is_accept())
{
    // If no, where do we first differ?
    int letter;
    differ_at = aut.num_states();
    int test_differ;
    int next1, next2;
    for (letter = 1; letter≤aut.num_letters(); letter++)
    {
        next1 = aut[s1][letter];
        next2 = aut[s2][letter];
        // Only go down a level if the next states do differ
        // Ones we're already looking at will return a value that's too large
        if (next1 ≠ next2)
        {
            test_differ = differ_depth(aut, next1, next2, first_differ);
            if (test_differ < differ_at) differ_at = test_differ;
        }
    }
    // Remember we're one higher up the tree
    differ_at++;
}

```

```

    }
    // Only store the answer if it's possible to evaluate it without looping
    // - otherwise forget we ever came this way
    if (differ_at  $\neq$  aut.num_states()+1)
    {
        first_differ.set(s1, s2, differ_at);
        first_differ.set(s2, s1, differ_at);
    }
    else
    {
        first_differ.set(s1, s2, -1);
        first_differ.set(s2, s1, -1);
    }
    return differ_at;
}

```


// Example 3: The guessing algorithm

//

```
void make_hypothesis(automaton& guess, const automaton& perfect, int first_dubious_word)
{
```

```
    table first_differ(perfect.num_states());
    intarray drop_down;
    intarray guess_states;
    intarray map_to;
    drop_down.set_length(perfect.num_states());
    drop_down.set_base(1);
    map_to.set_length(perfect.num_states());
    map_to.set_base(1);
    guess_states.set_length(perfect.num_states());
    guess_states.set_base(1);
```

```
    drop_down.zero_all();
    guess_states.zero_all();
    map_to.zero_all();
    guess = perfect;
    guess.clear();
    int max_sure_depth = first_dubious_word - 1;
    intarray todo;
```

```
    int pstate, cmpstate, pdepth;
    int deepest = 0;
    int warn_flag = 0;
    int j, k;
    todo[0] = 1;
```

```
    while (todo.len())
```

```
    {
        dout << "todo = " << todo << endl;
        pstate = todo[0];
        intarray temp = todo;
        todo = temp.mid(1);
        pdepth = perfect[pstate].depth();
        if (pdepth > deepest)
        {
            deepest = pdepth;
            dout << "Now at depth " << deepest << endl;
        }
        drop_down[pstate] = 0;
        if (pdepth > max_sure_depth) break;
        for (cmpstate=1; cmpstate<pstate; cmpstate++)
        {
            if (!drop_down[cmpstate])
            {
                if (differ_depth(perfect, cmpstate, pstate, first_differ)
                    + pdepth > max_sure_depth)
                {
                    if (drop_down[pstate])
                    {
                        if (!warn_flag)
                            cout << endl << "State " << pstate
                                << " indistinguishable from "
                                << drop_down[pstate] << " & " << cmpstate
                                << " which are themselves distinguishable"
```

```

        warn_flag = 1;
    }
    else
    {
        drop_down[pstate] = cmpstate;
        dout << pstate << " is indistinguishable from "
            << cmpstate << endl;
    }
}
}
if (!drop_down[pstate])
{
    guess.new_state();
    guess_states[guess.num_states()] = pstate;
    map_to[pstate] = guess.num_states();
    cout << pstate << "." << flush;
    dout << pstate << " survives as " << guess.num_states() << endl;
    // Add all of its children to the todo list
    for (j=1; j<=perfect.num_letters(); j++)
    {
        int child = perfect[pstate][j];
        // Only add to todo if we haven't seen it before
        // and it's not on the todo list already
        if (!drop_down[child] && !map_to[child])
        {
            for (k=0; k<todo.len(); k++)
            {
                if (todo[k] == child) break;
            }
            if (k == todo.len())
            {
                todo[todo.len()] = child;
                dout << "Must do " << child << endl;
            }
        }
    }
}
}
cout << endl;
if (pdepth <= max_sure_depth)
{
    cout << "Found terminal ring at depth " << deepest << endl;
    int gstate, letter, next_pstate;
    for (gstate = 1; gstate<=guess.num_states(); gstate++)
    {
        pstate = guess_states[gstate];
        if (perfect[pstate].is_accept())
        {
            guess[gstate].set_accept();
        }
        for (letter = 1; letter<=guess.num_letters(); letter++)
        {
            next_pstate = perfect[pstate][letter];
            if (drop_down[next_pstate])
                next_pstate = drop_down[next_pstate];

```

```

        guess[gstate][letter] = map_to[next_pstate];
    }
}
guess.sort();
cout << "Best guess: " << endl;
cout << guess;
int guess_rho = rho(guess);
int max_depth = guess[guess.num_states()].depth();
cout << "Max depth: " << max_depth << "; rho = "
    << guess_rho << "; fdw = " << first_dubious_word;
if (guess_rho + max_depth + 1 ≥ first_dubious_word)
    cout << " - doubtful";
cout << endl;
}
else
{
    cout << "Failed to get terminal ring" << endl;
    guess.clear();
    return;
}

intarray count;
guess.count_words(max_sure_depth, count);
if (!perfect.check_count(max_sure_depth, count))
{
    cout << "Fails word count with perfect" << endl;
    guess.clear();
    return;
}

if (perfect_contains_guess)
{
    if (!is_sublanguage(guess, perfect))
    {
        cout << "Guess not a sublanguage of perfect - forgetting." << endl;
        guess.clear();
    }
    else
    {
        cout << "Guess is sublanguage of perfect - OK!" << endl;
    }
}
else
{
    if (!is_sublanguage(perfect, guess))
    {
        cout << "Perfect not a sublanguage of guess - forgetting." << endl;
        guess.clear();
    }
    else
    {
        cout << "Perfect is sublanguage of guess - OK!" << endl;
    }
}
return;
}
}

```

Bibliography

- [AHU] A.V. Aho, J.E. Hopcroft & J.D. Ullman. *The design and analysis of computer algorithms*. Addison-Wesley, 1974.
- [CEHLPT] D.B.A. Epstein, J.W. Cannon, D.F. Holt, S.V.F. Levy, M.S. Paterson & W.P. Thurston. *Word processing in groups*. Jones and Bartlett, 1992.
- [EHR] D.B.A. Epstein, D.F. Holt & S.E. Rees. The use of Knuth-Bendix methods to solve the word problem in automatic groups. *Journal of Symbolic Computation*, 1991.
- [FK] R. Fricke & F. Klein. *Vorlesungen über die Theorie der automorphen Funktionen*. Teubner, Stuttgart, vol. 1, 1897; vol. 2, 1912.
- [GdelaH] E. Ghys & P. de la Harpe, editors. *Sur les groupes hyperboliques d'après Mikhael Gromov*. Birkhauser, Boston, 1990.
- [KB] D.E. Knuth & P.B. Bendix. Simple word problems in universal algebra. In J. Leech, editor, *Computational problems in abstract algebras*, pages 263–297. Pergamon Press, 1970.
- [KS] L. Keen & C. Series. Pleating coordinates for the Maskit Embedding of the Teichmüller Space of punctured tori. *Topology*, to appear.
- [Mag] W. Magnus. *Non-Euclidean tessellations and their groups*. Academic Press, 1974.
- [Mas] B. Maskit. Moduli of marked Riemann surfaces. *Bulletin of the A.M.S.* 80:773–777, 1974.
- [Mo] E.F. Moore. Gedanken experiments on sequential machines. In C.F. Shannon & J. McCarthy, editors, *Automata Studies* pages 129–153. Princeton University Press, 1956. *Annals of Mathematics Studies*, No. 34.
- [MPR] G. McShane, J.R. Parker & I.D. Redfern. *Drawing limit sets of Kleinian groups using finite state automata*. In preparation.
- [SH] R.E. Stearns & H.B. Hunt III. On the equivalence and containment problems for unambiguous regular expressions, regular grammars and finite automata. *SIAM Journal of Computing* 14:598–611, 1985.
- [TB] B.A. Trakhtenbrot & Y.M. Barzdin. *Finite automata – behavior and synthesis*. Translated from the Russian by D. Louvish; English translation edited by E. Shamir and L.H. Landweber. North-Holland, 1973. *Fundamental studies in computer science*, no. 1.
- [W] D.J. Wright. *The shape of the boundary of Maskit's embedding of the Teichmüller space of once-punctured tori*. Preprint from Oklahoma State University, 1988.